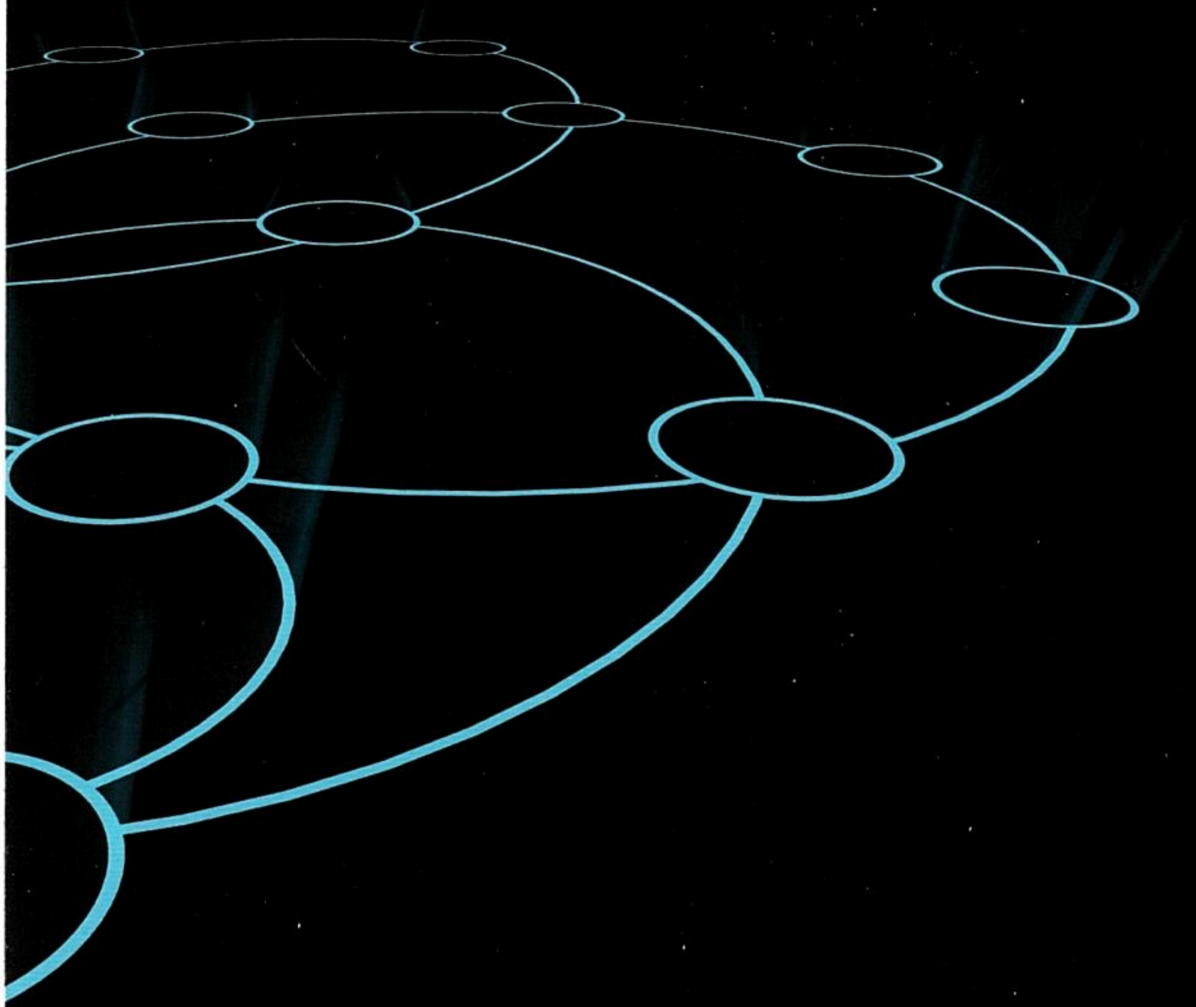
THE DATA CYCLOTRON

Juggling Data and Queries for a Data Warehouse Audience



Rómulo Gonçalves

The Data Cyclotron: Juggling Data and Queries for a Data Warehouse Audience

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Universiteit van Amsterdam
op gezag van de Rector Magnificus
prof. dr. D.C. van den Boom
ten overstaan van een door het college voor
promoties ingestelde commissie, in het openbaar
te verdedigen in de Aula der Universiteit
op vrijdag 22 maart 2013, te 11:00 uur

door

Rómulo António Pereira Gonçalves
geboren te Bolivar, Venezuela

Promotiecommissie

Promotor:	Prof. dr. M. L. Kersten
Overige leden:	Prof. dr. ir. C. T. A. M. de Laat
	Prof. dr. G. Alonso
	Dr. C. Mohan
	Prof. dr. P. Klint

Faculteit der Natuurwetenschappen, Wiskunde en Informatica

The research reported in this thesis has been partially carried out at *CWI*, the Dutch National Research Laboratory for Mathematics and Computer Science, within the theme *Database Architectures and Information Access*, a subdivision of the research cluster *Information Systems*.

The research reported in this thesis has been partially carried out as part of the continuous research and development of the MonetDB open-source database management system.

SIKS Dissertation Series No 2013-06.
The research reported in this thesis has been carried out under the auspices of *SIKS*, the Dutch Research School for Information and Knowledge Systems.

The research reported in this thesis was partially funded by NWO.

The cover was designed by Tiago Rosado [1].

The printing and the binding of this disseration was carried out by CPI Wöhrmann print service.

ISBN 978-90-6196-564-0

[1] tiago.m.rosado@gmail.com

Contents

Chapter 1

Introduction

1.1 A new Era

"The purpose of life is to obtain knowledge, use it to live with as much satisfaction as possible, and pass it on with improvements and modifications to the next generation", Apoorva Patel in [121]. Over billions of years, biological evolution has experimented a wide range of physical systems for acquiring, processing and communicating information [121].

The optimization principles behind those systems, i.e., minimization of errors and minimization of physical resources are a source of innovation for computing technology. Their goal has inspired and lead us to a new paradigm in Science, data-intense science [77]. A paradigm beyond experimental, theoretical research, and computer simulation.

From the world of science to the competitive world of Business Intelligence (BI) data has been collected, stored, and processed in any possible way. From simulation, experimentation, or simple logging of human and machine activity, the amount of data is growing exponentially. It is clear that information is becoming more relevant for our society evolution than ever. It has become known to the public as the*BigData Era*.

1.2 Challenges

Collecting data is only one step in a scientific investigation, and scientific knowledge is much more than a single compilation of data points. The world is full of observations to be made, but not every observation constitutes a useful piece of knowledge.

All scientists make choices about which data is most relevant to their research and what to do with it. In a continuous cycle, they try to turn a collection of measurements into a useful data set through analysis, and how to interpret it in the context of what they already know. The thoughtful and systematic collection, analysis, and interpretation of data allows a collection of measurements to be converted into evidence that supports scientific ideas, arguments, and hypotheses.

The process of data collection, analysis, and interpretation happens on multiple scales. It occurs over the course of a day, a year, or many years, and may involve one or many scientists whose priorities change over the time.

At the primary stages, huge amounts of raw data is collected to be scanned and filtered to remove noise or irrelevant properties. The filtering stage is followed by a simple aggregation phase to detect if the data is meaningfully or not. With a single scan simple conclusions are induced from this type of analyze. However, complex analysis requires several data scans to adjust the search parameters to fine tune model parameters.

The search moves from one database subset to another, zooming in and out until the complete data space has been explored. A good example is the search of a new galaxies in the universe. Using a catalog of the space, such as SkyServer catalog [137, 67, 82], astronomers request a region of the Universe to perform the search. On each region he extracts another sub-set or directly applies a complex and fine grain investigation. Each region is designed as context of interest.

The constant shift of focus leads to a short retention period for data- and workload-allocation decisions which may cause the resource utilization to deteriorate. These applications require a generic and simple data-access as well as flexibility to change the search space, reallocate resources on demand, and high throughput with modest response time.

Hence, a grand challenge for computer scientists is to devise a self-organizing architecture which exploits all hardware resources for the current workload, achieves an accurate database subset definition, minimizes response time, and maximizes throughput without a single point for global co-ordination.

We have accepted the challenge and in this dissertation we propose a novel architecture, the Data Cyclotron (DaCy), to exploit new trends in network hardware. It addresses the challenge using a turbulent data movement through a storage ring built from distributed main memory. Computations [1] assigned to individual nodes interact with the storage ring by picking up data fragments, which are continuously flowing around, i.e., the *hot-set*.

[1]A computation is a database query or a job defined by a user through the application integrated with Data Cyclotron.

The Data Cyclotron detaches itself from an old premise in distributed query processing, i.e., most distributed database systems from the past concentrate on reducing network cost. This was definitely a valid approach for the prevalent slow network connections. However, the continuous technology advancement calls for reconsideration of this design guideline.

Nevertheless, the rich research history for Database Management Systems (DBMSs) should not be ignored. On each Data Cyclotron node, a DBMS kernel is integrated to provide efficient techniques for data manipulation. With this processing efficiency and a symbiotic relationship with the network hardware trends, the Data Cyclotron stands in middle of state-of-the-art solutions as a novel approach for distributed data processing.

1.3 DataBase Management Systems history.

For more than three decades the manipulation of data stored in databases has been carried out by Database Management Systems (DBMS). The database term was introduced in mid-1960s at the same time direct-access storage for shared interactive use was made available [146].

The earliest DBMS, beyond efficiency, aimed to make the data independent from the logic of application programs, i.e., to split the logical schema from the storage schema, so the same data could be made available to different applications. An important feature to keep the user away from handling query plan optimization and resources allocation. A feature sometimes neglected by modern solutions, such as MapReduce, as we will later explain.

The DBMS had evolved from navigational systems, with Hierarchical and Codasyl model (Network model) as the most adopted ones, to relation systems. Due to the heavy demands on processing resources, relational systems only from the mid 1980s onward became widely deployed, i.e., computing hardware became powerful enough to sustain its heavy demands. Their dominance for all large-scale data processing applications became clear in the early 1990s and they still remain dominant in most areas.

A DBMS has evolved into a complex software system with the introduction of parallelism to load data, to build indexes, and to evaluate queries. Parallel DBMS improved query processing and data access latency by using multiple central processing units (CPUs) (including multi-core processors) and parallel storage. Contrary to sequential processing, with parallel processing operations are performed with overlap in time. Even in our days they still stand as one of the most efficient systems to scale vertically.

Though the parallel DBMS seek to improve performance in a single machine, i.e., scale up, they were not designed to scale out. Hence, distributed query processing was introduced through Distributed Database Management Systems (DDBMS) as early as the seventies.

The motivation for distributed query processing is to efficiently exploit a large resource pool: n nodes can potentially handle a larger workload more efficiently than a single node. The challenge is to make the queries meet the required data somewhere in the network, taking into account various parameters crucial in a distributed setting, e.g., network latency, load balancing, etc.

The state of the art for distributed query processing has evolved from static schemes over a limited number of processing nodes to architectures with no single point of failure, high resistance to network churn, flexible replication policies, efficient routing protocols, etc. [46, 99, 150].

1.4 State of the art

In the last decade several solutions have emerged to cope with new challenges for data warehouses, e.g., data extraction and online analysis on data sets that are growing into the multi-petabyte range. Most of these challenges are being met by high-performance data warehouses, Hadoop implementations, or data integration technologies. Each of them is designed to be efficient in one of the data processing stages, i.e., data extraction, data preparation, or data presentation.

The new solutions have started to bring advanced analytic computation closer and closer to the data. Such an approach is giving rise to in-database analytics. Predictive analysis, data mining, and other compute-intensive analytic functions have been migrated from separate, standalone applications into the enterprise data warehouse taking advantage of its full parallel-processing, scalability, and optimization features.

However, the solutions designed follow the old trends in network hardware. They were created for the inflexible tree structure used in data centers to optimize cost efficiency and access to specific outside data sources. Despite the improvements to adapt them to new data-center trends, their architectural design is still tied on two orthogonal, yet intertwined issues: data allocation and workload behavior.

In most architectures a node is made a priory responsible for a specific part of the database using, e.g., a key range or hash function. The query optimizer exploits the allocation function by contracting sub queries to specific nodes or issuing selective data movements and data replication between nodes. Unfortunately, the optimizer search space increases significantly too, making optimally exploitation of resources harder. This static data allocation calls for a predictable workload to advise optimal system configurations.

Workload behavior is, however, often not predictable. Therefore, an active query monitoring system is required, e.g., a database design wizard [34, 152, 103] to advice on indexes and materialized views, followed up with scheduled database maintenance actions. The problem is magnified in the presence of a skewed query workload in combination with a poor data allocation function. Furthermore, workload characteristics are not stable over time either. The workload continuously shifts from one part of the network to another making optimal data allocation scenarios extremely hard to find due to the high cost to reshuffle data kept on disks.

The ultimate goal is to design a self-organizing architecture, which maximizes resource utilization without global coordination, even in the presence of skewed and volatile workloads [34]. There is certainly room to design new data allocation functions [69], grid-base algorithms [25], distributed optimization techniques, and associated workload scheduling policies [107].

Several companies, e.g., Greenplum, Asterdata, Infobright, are designing new solutions to exploit large clusters and compute cloud infrastructures to increase the performance for business intelligence applications using modestly changed commodity open-source database systems. Even reduced, but scalable database functionality is entering the market, e.g., SimpleDB (Amazon) and Bigtable (Google).

However, it is clear that following a research track explored for three decades gets us only up to a certain point regarding improved resource utilization. We end up in a local optimum dictated by software inertia to follow the prime hardware trends such as increased CPU performance and disk bandwidth. It obscures the view of possible better approaches, if such a solution exists. Therefore, new network hardware trends should also be considered on the design of the optimal solution.

1.4.1 Hardware evolution

From sequencing genomes to monitoring the Earth's climate, from stock-market streams manipulation to search on social media many recent scientific and economical advances would not have been possible without data manipulation systems assisted with a parallel increase of computing power and network bandwidth.

The computing technology evolution is breaking old premises and assumptions used on the design of systems to cope with the data burst. New trends are emerging to manipulate and to explore massive data sets. Machines from one to multi-core processors, large main-memories, and high bandwidth for data-access are challenging the design of new ideas and reviving ideas once neglected.

In short-term view, network connections of 40 Gb/sec (3.7 GB/sec Ethernet) and

Remote Direct Memory Access (RDMA)[2] technology enables fast transfer of complete memory blocks from one machine to another in a single (fast) step. They are making massive processing, huge main memories, and fast interconnects affordable for experimentation with novel architectures [65]. Furthermore, thanks to network hardware road-map, the data-centers topology tends to be more flexible and heterogeneous. They are now built with intuition that customers constantly require different processing power and data access speed for each application they run.

1.5 The Data Cyclotron

In the spirit of an earlier attempt in the database field [75, 15], the *Data Cyclotron* architecture [63] was designed. It is a different approach for distributed query processing by reconsidering de-facto guidelines of the past [92]. It provides an outlook on a research landscape barely explored.

With the outlook of RDMA and fast interconnects, the Data Cyclotron is designed around processing nodes, comprised of state-of-the-art multi-core systems with sizable main memories, RDMA facilities, running an enhanced database management system, and a Data Cyclotron service.

1.5.1 Architecture

The Data Cyclotron service organizes the processing nodes into a *virtual storage ring* to send the database *hot-set* continuously around [3]. A query can be executed at any node on the ring; all it has to do is to announce its data interest and to wait for the data to pass by.

This way, continuous data movement becomes the Data Cyclotron key architectural characteristic. It is what database developers have been trying to avoid since the dawn of distributed query processing. It immediately raises numerous concerns and objections, which are better called research challenges. The query response time, query throughput, network latency, network management, routing, etc., are all areas that call for a re-evaluation.

We argue that hardware trends call for such a fresh, but unorthodox look and match it with a thorough assessment of the opportunities they create. Hence, the Data Cyclotron uses continuous data movement as the pivot for a continuous self-organizing system with load balancing.

[2]`http://www.rdmaconsortium.org/`

[3]For convenience, and without loss of generality, we assume that all data flows in the same direction, say *clockwise*.

1.5.2 Hot-Set management

Adaptation to changes in the workload is often a complex, expensive, and slow procedure. With the nodes tightly bound to a given data set, we first need to identify the workload change and the new workload pattern, then assign the new responsibilities, move the proper data to the proper nodes, and let everyone know about the new distribution.

In the Data Cyclotron a workload change affects the *hot-set* data on the ring, which is triggered by query requests for data access. Its self-organization in a distributed manner, keeping an optimal resource utilization, replaces gradually the data on the ring to accommodate to the current workload. The data, depending on its relevance for the workload, is kept in one of the three data sets, the *cold-set* (data in disk), *warm-set* (data in memory, but not in circulation), and the *hot-set* (data in circulation).

The integration of an innovative cache management reduces the number of times a data chunk commutes between the different data sets. Hence, high query throughput and low query response time is assured.

1.5.3 Efficient computation

For efficient computation the Data Cyclotron uses a main-memory column-store, known for its I/O efficiency. Efficient algorithms from the DBMS kernel are used for data processing. External libraries can also be integrated to explore different models of parallelization such as the MapReduce model for multi-core machines [88, 125].

Before compilation a computation [4] searches a lightly loaded node to execute on; the data needed will pass by upon request. This way, the load is not spread based on data assignment, but purely on the node's resources and on the storage ring load characteristics. This innovative and simple strategy intends to avoid *hot spots* that result from errors in the data allocation and computation plan algorithms.

Computations are compiled into an acyclic graph. Without much effort, the acyclic graph is split into independent sub-graphs to consume disjoint data subsets and exploit new multi-core processors to scale up. At execution time it is assured, by the Data Cyclotron, the data requested is available in memory.

1.5.4 Vision

The decentralized Data Cyclotron architecture allows us to have seamless transition between a scale up solution to a scale out solution, or the best case scenario, have both

[4] A computation is a query, or a job defined by a user through the application integrated with Data Cyclotron.

in harmony. Based on the resources of each unit, the DaCyDB has the flexibility to explore both models in different degrees to improve throughput and give response time guarantees.

With access to all data in all nodes, computation is split into sub-computations which are distributed among the nodes to explore distributed processing. They are sent to the ring and each of them settles on a different node following the basic procedures of a normal computation (cf., Section 1.5.3).

Sub-computations are processed concurrently and the individual intermediate results are then combined to form the final result. Such freedom gives the grounds for an application to scale out without be bounded to any complex scheduling algorithm or data distribution scheme.

Multi-computation processing can be boosted by reusing sub-computations result to avoid (part of) the processing cost, i.e., they are simply treated as persistent data and pushed into the storage ring for interested computations. Like base data, intermediate results keep flowing as long there is interest.

Based on the amount of computations, data flowing, and nodes load, the Data Cyclotron adapts the ring size to seek the optimal point for throughput. It uses a *pulsating ring*, one that adaptivaly *grows* and *shrinks* to find the optimal number of nodes comprising a ring and the optimal flow of data. A pulsating ring does not rely on a central coordinator, which makes the architecture flexible and robust to different workload resource demands. All decisions are independently done at each node. It is a new concept to design fully flexible distributed query processing architectures.

Moreover, it is conceivable that multiple pulsating rings live in the same physical cluster. At any point in time, such a *mesh* consists of *numerous* overlapped pulsating rings. Each ring focus on a given subset of the workload or database. Rings appear, grow/shrink, and disappear as the workload changes.

1.6 Synopsis

The scientific contributions of this dissertation have a dedicated Chapter each and they are all supported by publications in major international refereed venues.

Preceding each individual contribution, we present Chapter 2. This Chapter gives an overview of related concepts, and the background and contextualization for an independent read of the remain Chapters. It identifies data center trends followed by a short introduction on the state-of-the-art network hardware.

In all Chapters we have *anchors* to previous fractions of text to revive a concept or refresh the readers memory for a better contextualization of an idea.

" The anchors *are distinguished from the remain text as this paragraph is from the synopsis text. An* anchor *is preceded by the section identification and can be ignored in case the reader recalls the context. "*

1.6.1 Contributions

The scientific contributions of this dissertation can be summarized as follows:

1. The Data Cyclotron architecture, Chapter 3. The Chapter introduces the logical topology used by Data Cyclotron and its symbiotic relationship with current data centers and their new trends. Consequently, the three layers that compose a Data Cyclotron node [5], i.e., network layer, DaCy layer and application layer, are described in detail as well as their interaction. The Chapter ends with a description on how nodes interact to achieve high bandwidth and efficient memory utilization.

2. The Data Cyclotron *hot-set* management, Chapter 4. The Chapter presents and provides an in depth analysis, through simulation, of algorithms and protocols for an efficient *hot-set* definition. With uniform and non-uniform workload scenarios, we identify each *hot-set* management aspect that maximizes throughput and minimizes data access latency. Supported by observations collected during simulation, we propose optimizations and improvements to make the Data Cyclotron more adaptive to a diverged set of workloads and applications. Some proposals are then used and evaluated with a full functional system.

3. DBMS integration with the Data Cyclotron: DaCyDB, Chapter 5. The Chapter describes the conceptualization of a full functional system called DaCyDB. It shows the harmonious integration of a column-store DBMS with the Data Cyclotron. The integration and efficient parallelism is studied using two different clusters. Through micro-benchmarks and a well-known decision support benchmark, *TPC-H* [6], each layer in the architecture is tested for different amounts of resources. The Chapter concludes with a description of the features which equip DaCyDB with iterative data loading and different levels of data consistency.

4. Vision for DaCyDB, Chapter 6. The Chapter lays down the DaCyDB design details which make it an outstanding solution for distributed data analysis on huge data warehouses. The description goes from multi-query parallelization to scalable processing of complex queries and from pulsating rings for dynamic

[5]Also referred as *peer*.

[6]www.tpc.org

ring size adjustments to the co-existence of multi-rings within a single cluster. Furthermore, it positions DaCyDB with respect to other state-of-the-art systems for big data processing such as MapReduce frameworks emphasizing the advantages and disadvantages for each data processing phase. The Chapter concludes a thoughtful analyze of on going research for energy efficiency and describing how *green* is the DaCyDB architecture.

The dissertation is concluded with a wrap up of the main ideas and contributions in Chapter 7.

1.6.2 Publications

The work presented in this dissertation has been the basis for several publications in major international refereed database and distributed systems venues.

1. Romulo Goncalves, Martin Kersten. The Data Cyclotron Query Processing scheme. In the Proceedings of the 13th International Conference on Extending Database Technology (EDBT), pages 75-86, Lausanne, Switzerland, March 2010.

2. Philip W. Frey, Romulo Goncalves, Martin Kersten, Jens Teubner. A spinning join that does not get dizzy. In the Proceedings of the 30th International Conference on Distributed Computing Systems (ICDCS), pages 283-292, Genova, Italy, June 2010.

3. Philip W. Frey, Romulo Goncalves, Martin Kersten, Jens Teubner. Spinning relations: high-speed networks for distributed join processing. In the Proceedings of the 5th International Workshop on Data Management on New Hardware (DaMoN), pages 27-33, Providence, Rhode Island, USA, June 2009.

4. Romulo Goncalves, Martin Kersten. The Data Cyclotron query processing scheme. In ACM Transactions on Database Systems (TODS), volume 36, number 4, 27 pages , 2011.

5. Romulo Goncalves, Martin Kersten. An elastic system for distributed data analysis. Submitted for publication at the moment of printing this disseration.

Chapter 2

Background

The Data Cyclotron does not stand alone in the computer science research. The continuous flow of data using a ring topology have been studied for the last decades in several disciplines. In this Chapter we revisit the milestones influencing our architectural design and highlight related concepts in database research now ready for in-depth re-evaluation. Subsequently we discuss the driving force that is guiding the evolution of data centers and how the Data Cyclotron data-access protocols exploit the result of such evolution.

2.1 History

Distributed query processing to exploit remote memories and high speed networks to improve performance in OLTP has been studied in [80, 52]. It is also addressed in the area of distributed systems as data broadcasting and multi-cast messaging such as [75, 15, 4, 5, 7]. Solutions include scheduling techniques at the server side [4], caching techniques at the client side [4], integration of push-based and pull-based broadcasts [5], and pre-fetching techniques [7]. Most systems ignore the data content and their characteristics [96]. The seminal DataCycle [75] and Broadcast Disks approach [4] are exceptional systems to be looked at more carefully.

DataCycle.

The DataCycle project [75, 15] makes data items available by repetitive broadcast of the entire database stored using a central pump. The broadcasted database is filtered on-the-fly by microprocessors optimized for synchronous high-speed search, i.e., evaluation of all terms of a search predicate concurrently in the critical data movement path. It eliminates index creation and maintenance costs. The cycle time, i.e., the time to broadcast the entire database, is the major performance factor. It only depends on the speed of hardware components, the filter selectivity, and the network bandwidth.

The Data Cyclotron and DataCycle differ on four salient points. The Data Cyclotron uses a pull model to propagate only relevant data for the query workload, this is, the *hot-set*. Through requests, nodes express their interest and define the content of the *hot-set*. It does not distinguish the participants, all nodes access the data and contribute with data, i.e., there is no central pump. Each node bulk loads database content structured by a catalog, such as relational scheme, and not as a tuple stream. Finally, the Data Cyclotron performance relies on cache policies to remove irrelevant data from circulation and dynamic re-adjustment of the *hot-set* to exploit the available resources such as ring capacity and speed. This self-organizing behavior is the key for the success of our architecture compared to DataCycle.

A distributed version of DataCycle has been proposed in [27] and studied in [15]. The distributed scheme is a solution for the lack of scalability of the central scheme. Moreover, they focus on the data consistency when several pumps are used. The optimization of the broadcast set, to reduce the latency, is pointed out, but not studied in detail.

Broadcast Disks.

Broadcast Disks [4] superimposes multiple disks spinning at different speeds on a single broadcast channel creating an arbitrarily fine-grained memory hierarchy. It uses a novel multi-disk structuring mechanism that allows data items to be broadcasted non-uniformly, i.e., bandwidth can be allocated to data items in proportion to their importance. Furthermore, it provides client-side storage management algorithms for data caching and pre-fetching tailored to the multi-disk broadcast.

The Broadcast Disk is designed for asymmetric communication environments. The central pump broadcasts according to a periodic schedule in anticipation of client requests. In later work a pull-back channel was integrated to allow clients to send explicit requests for data to the server. However, it does not combine client requests to reduce the stress on the channel.

In [5] the authors address the threshold between the pull and the push approach. For a lightly loaded server the pull-based policy is the preferred one. However, the pure push-based policy works best on a saturated server. Using both techniques in a system with widely varying loads can lead to significant degradation of performance, since they fail to scale once the server load moves away from its optimal niche. The IPP algorithm, a merge between both extremes pull- and push-based algorithm, provided reasonably consistent performance over the entire spectrum of the system load.

The Data Cyclotron differs in two main aspects, a) it is not designed for asymmetric communication environments and b) it does not have a central pump. All nodes participate in the data and requests flow. Furthermore, requests are combined to reduce the upstream traffic. Therefore, we do not encounter problems using a pure pull model which according to [5] is suited for systems with dynamic loads. Finally, for data propagation, no replication on the data stream is used to create channels with different bandwidths, i.e, we do not have a multi-disk structuring mechanism.

2.2 Related concepts

In the literature there are two works which exploit a continuous data stream to boost query performance, *DataPath* [11] and *Cooperative Scans* [155]. *DataPath* explores a continuous data flow between the memory and the CPU Cores to amortize the data access costs among every computation that uses the data. *Cooperative Scans* synchronizes the queries to have a continuous stream of data from disk.

More than just share a data scan, the Data Cyclotron also shares work among the computations to improve throughput. A concept shared with ShareDB [62]. ShareDB batches queries to share computation across possible hundreds of concurrent queries and updates.

A further shared concept is the use of a group of inter-connected nodes to perform distributed query processing. Peer-2-peer and Grid architectures have attempted to create efficient and flexible solutions. Lately we have seen the raise of the MapReduce paradigm for Petabyte data sets.

DataPath.

In DataPath queries do not request data. Instead, data is automatically pushed onto processors where they are then processed by any interested computation. If there is no spare computation to process the data, the data is simply dropped. All query processing in the DataPath system is centered around the idea of a single, central path network. A DataPath's path network is a graph consisting of data streams (called paths) that force data into computations (called waypoints).

Contrary to Data Cyclotron, the DataPath is designed for a single machine. It is bounded by the local number of cores and the main-memory size, i.e., it cannot scale. Furthermore, it needs dozens of disks to optimize the data load into main-memory, i.e, the data partition and distribution is driven by the workload type.

Being a pure push-based model workload shifts, or queries with dissimilar plans, complicate the adaptation of the path network. Hence, the system is not as flexible as Data Cyclotron.

Cooperative scans.

Cooperative Scans [155] exploits the observation that concurrent scans have a high expected amount of overlapping data need. They explore the opportunity to synchronize queries, such that they share buffered data and thus reduce the demand for disk bandwidth. The synchronization uses a relevance policy to prioritize starved queries. It tries to maximize buffer pool reuse without slowing down fast queries, i.e., it maximizes throughput and minimizes latency.

The queries are prioritized according to the amount of data they still need, with shorter queries receiving higher priorities. Furthermore, to maximize sharing, chunks that are needed by the highest number of starved queries are promoted, while at the same time, the priorities of chunks needed by many non-starved queries are adjusted slightly.

The Data Cyclotron and Cooperative Scans share the goal to improve throughput and minimize latency. However, the Data Cyclotron was designed for distributed query processing and explores the access to remote memory instead of to disk. Although, some of the techniques and observations made in [155] may still be applicable to the Data Cyclotron.

ShareDB.

ShareDB is based on a new processing model that batches queries and updates in order to share computation across them [62]. It detaches itself from the traditional query-at-the-time processing model by compiling hundreds of queries into a single plan, i.e., the operators and input data are shared among all queries in the batch. For example, instead of doing N independent partial joins between two relations, a single big join is executed and the result routed to each query.

The cumulative cost increase to answer the first queries added to the batch and the extra work compared to a traditional query-at-the-time approach, at a first glance, seems to be the *show stopper* of this elegant processing model. However, with hundreds of partial joins between the same relation there will be a significant overlap between the intermediates for those queries. Hence, the more queries, the more the overlap, and therefore, a higher throughput.

ShareDB is designed to handle high throughput with response time guarantees on highly complex and dynamic workloads such as the ones represented by the TPC-W benchmark. The DaCyDB shares the same goal as ShareDB, but it takes a different path to share work among the computations. DaCyDB uses a divide and conquer approach to split huge and complex computations into smaller ones and detect commonalities among them. Through the instructions and sub-functions result materialization, DaCyDB boosts multi-query processing by reusing them, and thus avoid (part of) the processing cost. Furthermore, it is aimed for workloads similar to a TPC-H workload.

Nevertheless, they have several concepts in common and the ShareDB techniques to define the global plan could be considered to reduce the amount of data overlapping among the flowing intermediates.

The peer-2-peer and Grid architectures.

A more recent attempt to harness the potentials of distributed processing are the peer-2-peer and Grid architectures. They are built on the assumption that *a node is statically bound to a specific part of the data space*. This assumption is necessary to create a manageable structure for query optimizers to derive plans for queries and data to meet. However, a static assignment also complicates reaching satisfactory load balancing, in particular with dynamic workloads, i.e., at any time *only part of the network is being used*.

For example, in a Distributed Hash Table (DHT), such as [126, 145], each node is responsible for all queries and tuples whose hash-values fall within a given range. The position of a node in the overlay ring defines this range. This is a significantly more dynamic approach as a node can, in a quick and inexpensive way, change its position in the DHT. It effectively transfers part of its load to another node. However, even in these architectures, at any time, a node (or set of nodes) is still responsible only for part of the hash-value domain or data set. With continuous change of focus, the response time is taxed by the continuous readjustments to the DHT.

The main difference with the Data Cyclotron is our reliance on a full fledged database engine to solve the queries, and shipping large data fragments around rather then individual tuples selected by their hash value.

The MapReduce model.

The MapReduce pattern, better known as the Master-Worker pattern, is used for parallel processing. It follows a simple approach that allows applications to perform simultaneous processing across multiple machines or processes via a master and multiple workers, see Figure 2.1. Its popularity is derived from the growing dominance of web applications and cloud computing. Both of them have architectures that require scale-out solutions.

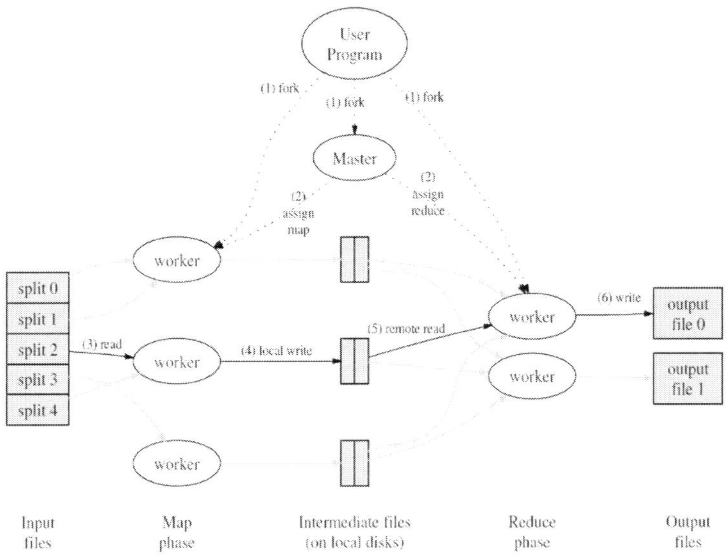


Figure 2.1: Execution overview [45].

MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a map function that processes a key/value pair list to generate a set of intermediate key/value pairs list, and a reduce function that merges all intermediate values associated with the same intermediate key [45].

All intermediate values associated with the same intermediate key K are grouped and passed to the reduce function. The reduce function accepts a set of values for an intermediate key K and merges them to form a possibly smaller set of values. Typically, just zero or one output value is produced per reduce invocation. The intermediate values are supplied to the user's reduce function via an iterator so lists of values that are too large to fit in memory can be handled.

The abstraction is inspired by the map and reduce primitives present in *Lisp* and many other functional languages. The use of a functional model with user specified map and reduce operations allows any MapReduce framework to parallelize large computations easily and to use re-execution as the primary mechanism for fault tolerance.

MapReduce differs from Data Cyclotron in two major aspects, parallelization and data shuffling. The parallelization is done through two independent phases, map and reduce, which are executed one after the other. The data shuffle uses 2D mesh approach, i.e., M mappers send data to R reducers ($M \times R$ connections). The Data Cyclotron does not use any predefined phase and data shuffling is done through a continuous stream using ring topology.

Several MapReduce inspired frameworks have been design in the last years. In the coming Section 2.3 the most novel solutions are presented and their similarities and differences with the Data Cyclotron architecture discussed.

2.3 MapReduce frameworks

Several solutions, such as high-performance data warehouses, Hadoop implementations, extreme performance, or data integration technologies, have emerged to cope with new challenges to manage and extract information from databases that are growing into the multi-petabyte range. For example, retailers are enriching the information in their customer data warehouse with years of click-level web activity for each customer. Banks are reducing data latency for their production loan assessment and approval applications from intra-day to sub-one hour latency [108]. Scientists are exploring new worlds with the famous fourth paradigm, and thus collecting Terabytes of experimental results for further searching of the unknown.

The most adopted solutions are the new MapReduce-inspired massive data processing platforms. Dryad [81], Hyracks [24], Spark [151], Nephele [143], Ciel [114], all include elements of MapReduce, but with more choices in runtime query execution. Nevertheless, the essence underlying these parallel programming frameworks is that they are all derived from a multi-phase parallel programming model which is known for its inefficient resource utilization [49].

In this Section we dissect a well known MapReduce framework, Hadoop, and its derivations to understand the advantages and disadvantages of its architecture for certain workload scenarios. From data distribution to resource utilization, from data processing to integration with a DBMS, the major features are presented as well as the latest solutions to improve them. The analyze intends to bring out the most important architectural design features of MapReduce-inspired solutions which do not make them suitable for complex data analyzes.

2.3.1 Data distribution

A key feature of Map Reduce is the direct use of data as it is stored, i.e., it does not require database pre-loading. For DBMS users, loading all data is a huge investment to let queries run in an efficient way. However, when the user only runs few queries over

Petabyte size data sets, the user does not gain anything from the investment of a priory loading of all data. The idea is now being explored in the database community such as Oracle through the external tables [119] feature, *lazy* ETL [90], or NoDB [9].

Most Map Reduce frameworks, such as Hadoop [71], access the data directly from a distributed file system. Hadoop distribute file system (HDFS) is widely used for ELT systems to provide fault tolerance and easy data access. The user can run multiple jobs directly over the data stored in the distribute file system. The files are split and distributed in such a way that computations can be pushed into the nodes keeping the data shuffling to the minimum. Hence, computations are defined over the data as it is without pre-loading it into specific structure or scheme.

At data distribution time, files are split into blocks of 64 MB size and they are distributed among the nodes through (by default) three replicas. Two within the same rack to explore locality [1], and the third one in another rack to be robust against rack failures. The partitioning aims for a uniform distribution of load and to reduce as much as possible data shuffling, i.e., computation is moved to the nodes to avoid data movement [148].

The approach is efficient if the workloads are uniform and the partitioning function aligns with the blocks distribution, i.e., each partition is composed of one or more blocks resident in the same node or, in worst case, in a node within the same rack. Otherwise, performance is negatively affected by data shuffling between nodes which is taxed by distributed file system synchronization of dirty pages and load at the source node [44]. Upon a shift on the workload focus, the data has to be all re-distributed.

Using a distributed file system has also another drawback. Sometimes all or many of the tasks in a MapReduce job list need to access a single file or a set of files. For example, when joining a large file with a small file the predominant approach is to open the small file as a side file. The small file is opened directly in the map task rather than be specified as an input to the MapReduce job. It is opened, loaded into memory and used for the join in the map phase [59].

When thousands of map or reduce tasks attempt to open the same HDFS file simultaneously a large strain occurs on the *NameNode* and the *DataNodes* storing that file [59]. To avoid this situation MapReduce provides a distributed cache [70]. The distributed cache allows users to specify any HDFS file they want every task to have access to. These files are copied into the local disk of the task nodes. The approach adds extra overhead before the job can be initialized and the data might need to be re-shuffled again when different jobs are submitted.

[1] It is assumed the latency to access data in the local disk is the same as the one to access data in remote node within the same rack.

2.3.2 Resource utilization

For simple computations over large data sets, MapReduce frameworks have a balanced resource utilization. These computations tend to be only I/O bounded. Therefore, a number of partitions, i.e., mappers, higher than the number of map slots improves I/O and reduces skewness. The allocated slots are used in equal manner due to the high number of mappers per map slot. Furthermore, the computation simplicity contributes for a small difference between map phase and reduce phase.

For complex computations MapReduce solutions looses resource utilization efficiency. The inefficiency is due to three drawbacks, single node resource usage unbalance, reduce slot hoarding, and resource allocation unbalance within a job [142].

First, within a round of map tasks, or reduce tasks, different phases have varying resource usage. With static allocation without considering the node's load, homogeneous tasks which execute at the same pace might have some resources, such IO or CPU, under-used while others over-used [142].

Second, each reducer copies its portion of the results of each map, and can only apply the user's reduce function once it has results from all mappers. For a big job, with a lengthy map phase, the reduce slots are hold up until all map tasks are finished, this produces underutilized resources.

Last, resources are allocated by a static configuration which does not consider the system dynamical load and job requirements. The number of slots is always fixed independently of the number of mappers or reducers. Furthermore, the cluster might have too many map slots, but not enough reduce slots, and vice-versa.

Extensions to Hadoop have been proposed to overcome this resource allocation issue. In [142] authors have proposed a solution that estimates the number of mappers and reducers to avoid the resource allocation unbalance problem. To improve resource utilization the map execution, and also the reduce execution, is split into two phases, CPU and I/O phase. The task scheduling attempts then to overlap their different phases. For example, when a reduce task enters into sort-compute period, the the IO period of the latter reduce task is overlapped with the CPU period of the former reduce task [142].

The solution is promising but it is also bounded with two issues. It is too dependent on static configuration and it is compromised by the network CPU overload. The CPU is affected by network transfers which makes the scheduling of each phase complicate. Despite that, the allocation continues to be inflexible when the workload changes or only a sub-set of the data is used.

2.3.3 Data processing

Hadoop is *data-parallel*, but *process-sequential*. Within a job, parallelism happens within a map phase and a reduce phase. These two phases, however, cannot be run in parallel. The reduce phase cannot be started until the map phase is fully completed [78].

Applications that need to scan the entire data-set to do aggregations, or collect statistics, are perfect for a Map-Reduce approach. However, random ad/hoc processing of a small subset of data within a large data set is not a suitable scenario for Hadoop [78]. Inefficient resource utilization increases the MapReduce response time. The major reason is the lack of adaptive and cooperative data access by independent jobs.

It is known that adaptive algorithms have superior performance stability as they are robust to tuning errors and changing runtime conditions such as other jobs running on the same cluster. For Hadoop, the adaptive behavior was introduced with *Adaptive MapReduce* [141].

The authors in [141] decoupled the number of splits (partitions) from the number of mappers to obtain the best of both worlds. That is, load balancing, reduced scheduling and starting overhead, and combiner benefit. The solution made Hadoop more flexible and released the user from hard performance tuning [12].

To achieve it, authors made possible that more than one split can be assigned to a single mapper. After all data from the chosen split is processed, the mapper tries to pick a new split. If it succeeds, mapper processing continues unaffected. The split switch is transparent to the map function unless it explicitly asks for the split location. When mappers finish processing local splits, they start processing any unprocessed remote splits. If no more splits are available, mappers end their execution. Once an adaptive mapper stops, the reducers can start copying its output [141].

All the management of the assigned and unassigned splits is done with Zookeeper [153]. It allows all sorts of synchronization such as locks or queues when the task are running in parallel. However, every synchronization is a bottleneck and restricts the workflows from scaling [18]. The dependency on a central coordination point, the meta-data store, even if it is a decentralized structure and fault-tolerant, reduces the flexibility from the decentralized line of thinking of MapReduce. Despite that, the idea would have a bigger impact if the adeptness was also for inter-task and not only for intra-task operations.

2.3.4 Complex computations

The data processing model for new MapReduce solutions have proved to be more adaptive and dynamic. Despite the improvements, their architecture is still considered a *space based architecture*. With a space-based architecture, applications are built out of a set of self-sufficient units, known as processing-units (PU). These units are independent of each other, so that the application can scale by adding more units [147].

The evolution of MapReduce-inspired solutions is predominant composed of scale out architectures. The resources in that architectures are getting bigger and bigger to manage more data without increasing the complexity of managing it [130]. To maximize the use of these bigger resources, a scale-up approach needs to be combined as well. It brings MapReduce frameworks close to the efficiency levels of parallel DBMSs.

An initial comparison of MapReduce and parallel DBMSs [133] pointed out a substantial performance difference between the two platforms. Recent studies for MapReduce have demonstrated that indexing [47, 85], columnar storage [51], and other organization techniques [87] can be supported by MapReduce to reduce this performance gap. Other approaches [19, 6, 149] have discussed join processing on MapReduce. In [76] the authors have examined how to tune the various parameters in a MapReduce platform like Hadoop to provide the best performance for a given job [89].

The authors in [89] showed that these efforts address general workloads and can not particularly be focused on structured data such as the star schema. Hence, they built a prototype, called Clydesdale, to assemble all these techniques to Hadoop to improve resource utilization and performance for start schema databases. Clydesdale draws from columnar storage [51] for I/O performance, join techniques [1] suitable for multi-core servers and Hadoop, and block-iteration [154] for CPU performance.

It uses carefully designed map tasks so that data structures used for query processing are shared across multiple threads and even multiple tasks consecutively executed on any node. Such design allows Clydesdale to inherit the fault-tolerance, elasticity, and scalability properties of MapReduce. Using the star schema benchmark [118], the authors have shown they are five times to eighty three times faster than Hive [139].

It is a promising research prototype, but not a fully functional system like Hive or DaCyDB. It does not currently have a SQL parser, queries are written in Java as MapReduce programs. Using Java, Clydesdale is exposed to the Java Virtual Memory garbage collector issues when it wants to exploit servers with large memories and multiple cores for large data sets analyses.

An asynchronous garbage collection happens when big heaps are allocated. It happens at unpredictable intervals and stops the Java program from executing until it is completed. This garbage collection pauses are often considered the *Achilles Heel* of Java because of the poor guaranteed response or query times that may result. It also makes the application inflexible to handle changing workloads. Hence, scalability for in-memory computing, the right approach for real-time big data analysis and large start schema data sets, is not possible with this type of garbage collection [129].

Furthermore, using Hadoop distributed file system (HDFS) for structured data processing, its response time is compromised by the major HDFS bottleneck, bandwidth. The authors [89] have spotted that the bandwidth at which map tasks could read from HDFS was only a fraction of the raw disk bandwidth. For instance, they measured that each disk was able to supply between 70 MB/s and 100 MB/s. Conservatively assuming 70 MB/s per disk would result in 560 MB/sec for eight disks and 280 MB/sec in case of four disks, benchmark results shown 126 MB/sec for eight disks and 105 MB/sec for four disks

2.3.5 Using a DBMS

The disadvantages of using distributed file system can be cut off through the integration of a DBMS instance in each node. This radical change to the MapReduce platform was successfully done on HadoopDB [2]. HadoopDB tries to combine the flexibility and scalability of MapReduce with the performance of a relational DBMS.

HadoopDB [2] discards the distributed file system in favor of individual database nodes to construct a parallel DBMS. It combines a single node relational database with Hadoop to get better performance. Reliable storage or replication is used to have data fault tolerance.

The initial approach [2] has integrated a row-store, PostGres. The later version they have scale up their solution to use a column-store DBMS, VectorWise (VW) which has vectorized operations on in-cache data and efficient I/O [3].

The performance gains compared to Hive and a commercial parallel database are impressive. They have shown that the efficient processing capacities of a DBMS and the drop of the distributed file system has brought another degree of performance never achieved before by all Hadoop extensions.

Nevertheless, they are still dependent of the underline network layer of MapReduce which is inefficient and compromises the HadoopDB project success. Randolph stated in [112] that early DeepCloud [39] prototypes [2] have shown the drawbacks of using MapReduce primitives as the main distribution mechanism [3]. They observed that a DBMS node that was 70 times faster than using HDFS, just made the data movement system look really slow.

MapReduce has latency problems that are both inherent in its batch nature and its use of TCP. TCP typically increases the communications latency by at least a factor of 10. TCP round trip messaging times are as high as 400 ms. Contrary, OpenFabrics/InfiniBand round trips do this in less than 4ms. Such propagation delays become

[2]http://deepcloud.co/web1/deepcloud-6-pack-is-now-available-for-download/

[3]http://deepcloud.co/web1/its-official-mapreduce-is-slow/

much more important as the cluster becomes larger. Hence, they discarded the option of using MapReduce primitives as they also dropped TCP in favor of RDMA when they started using VectorWise nodes.

The adoption of RDMA by DeepCloud confirms that the strategy taken by Data Cyclotron in exploring the new trends on network hardware is the right one. The Hadoop community has realized that and the first step has been taken by Mellanox UDA [109]. A novel data moving protocol which uses RDMA in combination with an efficient merge-sort algorithm. It enables Hadoop clusters based on Mellanox InfiniBand and 10 Gb RoCE (RDMA over Converged Ethernet) adapter cards to efficiently move data between servers. However, contrary to the Data Cyclotron, the underlying network model used by MapReduce [4] is not suitable to exploit all the capacities of RDMA [55] (detailed discussion in Section 2.6.4).

2.4 Data center trends

The number of data centers constructed by enterprises has sky rocked over the last decade. In the same line are the data-sets size, servers, and storage devices requesting network expansion. Moreover, enterprises are encountering increasing pressure in terms of the rational use of space and power consumption.

With the network as the key service bearer of the data center, network infrastructure is facing continuous challenges. The network technologies improvement to cope with them is guiding data centers evolution.

2.4.1 Driving force

The driving force of data centers evolution is demanded by high concentration of high-density application systems. They request the state of art in physical servers and storage units. The growth of processing capacity complies with Moore's Law and strengthens the data centers processing capacity. However, they demand high bandwidths for data access.

Furthermore, virtualization is also a trend in most of the data-centers where a single and high-capacity physical server is virtualized into several logical computing units, raising the computing capacity and access to storage units to a higher level. The network connects all of them. It is the backplane for the virtual servers.

With several application systems of multiple logical servers piled onto a network interface, the data center infrastructure faces performance requirements that are several

[4]The data shuffle uses 2D mesh approach, i.e., M mappers send data to R reducers ($M \times R$ connections).

times larger than a traditional system. In addition, a highly reliable infrastructure is indispensable in such heavy application environment.

The high dense application environment is increasing the service traffic and introducing traffic bursts [5]. The switching system tries to buffer and queue traffic bursts, however, the high ratios between each network level creates severe congestion increasing packet loss. A packet loss is not noticed at application level, but it indirectly affects the application performance due to the increase of data access latency [68].

2.4.2 Data centers evolution

The evolution is guiding the data centers to an unified switching network that can effectively support front end service access, high-speed server interconnections, and access to storage units.

A data center is now deployed with multiple independent networks to address different application connection requirements. Storage access networks (SANs) for the access of upper-layer application system on servers to back end storage units. Switching networks, Ethernet or InfiniBand, for reliable, secure, and remote connectivity to end user's. High-speed server interconnection networks for high-speed server clusters [6].

Structural organization.

A server is not anymore a single box with all compute resources in it. They have been placed in pools together with storage and memory processors. All components have been spread around the data center and they are all interconnected through a network.

Cloud computing platforms such as Windows Azure, Yahoo storage, or Google storage, have their storage services separated from compute nodes. To bring the storage network and compute server groups together, the tendency is to use storage area networks (SANs). They use a fiber channel over Ethernet ($FCoE$) or InfiniBand to interconnect high-performance servers and storage nodes at the core of the data centers.

$FCoE$ maps Fiber Channel (FC) directly over Ethernet while being independent of the Ethernet forwarding scheme. By retaining the native FC constructs, $FCoE$ was meant to integrate with existing FC networks and management software. Using FC rather then TCP/IP, i.e., it encapsulates FC packets over the native Ethernet transport, $FCoE$ ensures loss-less transmission of data.

[5]According to observations, within a 1ms, a peek traffic value 2 or 3 times the average value [68].

[6]In most cases, Ethernet networks are used for services and InfiniBand (IB) networks are used for cluster interconnections.

It is truth that a major part of the serves still use direct-attached storage (DAS) for temporary storage, such as caching or logging for fault-tolerance, but the use of diskless nodes is on the horizon.

Network technologies.

Ethernet technology development has been multiplying its speeds by a factor 10 over the last decade [68] [7]. With the costs continuously decreasing, the 10 Gb/sec structure (or higher) begins to move from the core to the network edge, servers, and storage units. For example, the next generation of Ethernet standards has a single standard covering 40 Gb/sec and 100 Gb/sec [8].

InfiniBand (IB) is more mature and more widely used in high-performance computing environments because it is more economical and already has higher bandwidth capabilities than Ethernet. Furthermore, InfiniBand has Remote Direct Memory Access (RDMA) and is open source. With RDMA, InfiniBand is more *green* since RDMA consumes less energy. A more detail discussion on InfiniBand and RDMA technology is presented in Section 2.5 and Section 2.6.

All these upgrades are small increments and they have primarily been speed upgrades not architecture upgrades. A re-think of network architectures is now promoting the adoption of new topologies.

Network topologies.

Tree topology has been used for several years in data centers. Those data centers were built without high communication complexity computations in mind. They were mainly optimized for scalable access by external clients. Over time, the tree topology evolved from a binary or ternary tree to a flat tree or a star topology to coupe with the increase of communication complexity.

The major disadvantage of the initial topology was the congestion at the upper nodes if several processors further down wish to communicate. One way to alleviate this communication problem was to add additional connections between levels, i.e., by increasing the bisection bandwidth. This type of tree is called fat tree.

The edges of a fat tree are most of the times organized as a star. A star topology is a tree of depth two and usually has a high fan out [9]. It has growth complexity one since only the central node has to be modified to cope with the extra node. The central

[7]100 Mb/sec in 1995, 1 Gb/sec in 1998, 10 Gb/sec in 2002, and 100 Gb/sec in 2010.

[8]IEEE802.3ba., the 40 Gb/sec rate is oriented mainly for servers and the 100 Gb/sec for convergence and backbone layers.

[9]The fan-out represents the number of nodes to each a single node is directly connected.

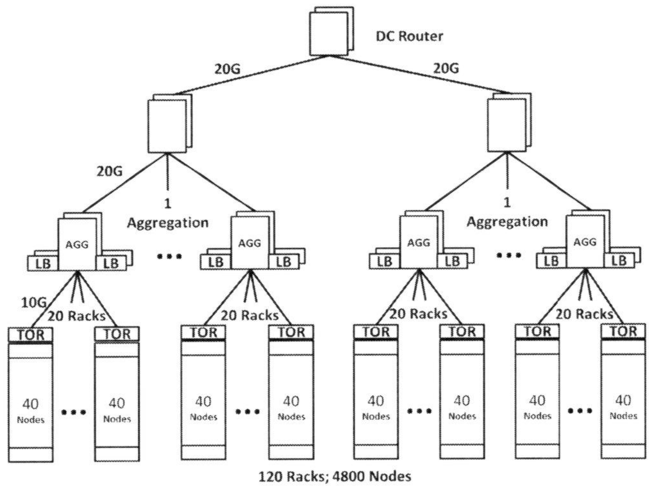


Figure 2.2: Fat tree topology used in current data centers.

node is thus a potential communication bottleneck. Consequently star network tends to be heterogeneous with the central processor being designed with a much higher communication bandwidth than the others such as a top rack switch. An example of this type of organization is picture in Figure 2.2.

New decentralized topologies are emerging for more efficient communication between nodes such as full or partial mesh topology. A fully connected network, or full mesh topology, is a network topology in which there is a direct link between all pairs of nodes. In a fully connected network with N nodes, there are $N(N-1)/2$ direct links. This type of topology, despite its high degree of reliability, are not used in big data centers since they are too expensive to set up.

In a partial mesh topology there are at least two nodes with two or more paths between them to provide redundant paths. This decentralization is often used to compensate for the single-point-failure disadvantage that is present when using a single device as a central node, e.g., star topology and tree topology.

A special kind of partial mesh, limiting the number of hops between two nodes, is a hypercube. Hypercube assures high bandwidth, low latency, and it can embed logical topologies such as mesh, tree, etc. However, it is costly and hard to build due to the higher number links required to connect all the nodes. The degree of complexity increases every time it needs to scale. The number of I/O ports increases and the cube-

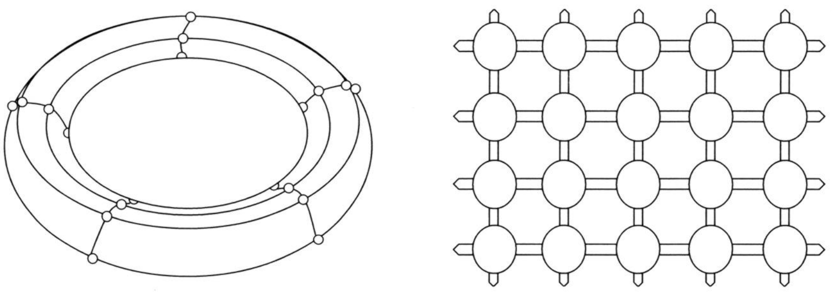

(a) Hierarchical ring. (b) Torus 2D.

Figure 2.3: Multi-dimensional ring topologies.

connected cycles as well. As alternative a grid network where a linear, or ring, topology is used to connect systems in multiple directions, for instance, a multi-dimensional ring as represented in Figure 2.3. They are known for their decentralized nature.

Recent developments in the research world are enforcing new radical changes on the design of network topologies. On the horizon, the network in a data center will be like a cloud where nodes are randomly connected, or not, to several other nodes. The network will become more symmetric allowing the computational image of a node to migrate around the data center or be share between different group of nodes [8].

2.5 InfiniBand technology

The Data Cyclotron is designed for both Ethernet and InfiniBand. However, InfiniBand is preferred for its better technology in latency and bandwidth management. Moreover, it does not put limitations on the logical topologies to be implemented and it makes efficient use of the underlying physical topology.

InfiniBand is a multi-lane protocol. Generally speaking, the four-lane (4x) products are used to link servers to switches, the eight-lane (8x) products are used for switch uplinks, and the twelve-lane (12x) products are used for switch-to-switch links. The single-lane (1x) products are intended to run the InfiniBand protocol over wide area networks.

In 2000, the original InfiniBand ran each lane at 2.5 Gb/sec (single data rate). However, in 2005, the DDR (double data rate) pushed it up to 5 Gb/sec and since 2008, the

QDR (quad data rate) products ran each lane at 10 Gb/sec. Of 2011, InfiniBand has two new lane speeds using 64/66 encoding. The first is FDR (fourteen data rate) with 14 Gb/sec speed. The second is called EDR, short for Eight Data Rate, which actually runs the InfiniBand lanes at 25 Gb/sec. A bandwidth of 300 Gb/sec bandwidth between any two nodes using 12x EDR comes within reach [10].

With InfiniBand road map, the bottleneck in data exchange is shifting to the PCI Express bus. The PCI Express 2.0 technology, highly used on current data centers, is still based on old encoding techniques (8b/10b encoding) [11]. PCI Express 3.0 has recently been released [12] with new encoding (128b/130b) and higher bandwidth (64 Gb/sec instead of the 32 Gb/sec for PCI 2.0), but it sill can not catch up with the InfiniBand bandwidth evolution. InfiniBand has however a technology to reduce the memory BUS overhead for data transfers. This technology is called Remote Direct Memory Access (RDMA) and it is presented in detail in Section 2.6.

2.6 Remote Direct Memory Access

Network communication is known to be compute- and memory I/O-intensive [53]. High-volume data transfers thus depend on dedicated hardware assistance, which modern network cards provide in terms of *Remote Direct Memory Access (RDMA)*. Network interface cards (NICs) that provide RDMA functionality are also called *RDMA-enabled NICs* or *RNICs*.

To make efficient use of RDMA, however, applications have to respect some of the characteristics of the hardware-accelerated transport mechanism. This section summarizes the most relevant characteristics extracted from a RDMA evaluation [57]. They are the key features which motivate the design of the Data Cyclotron.

2.6.1 Traditional TCP communication

In traditional TCP network communication, typically implemented using the Berkeley `socket` API, the operating system kernel is in charge of processing the network stack which includes executing the respective protocols and moving the payload data.

The intensive involvement of the kernel on both sides causes a substantial amount of CPU load. A rule of thumb in network processing states that about 1 GHz in CPU performance is necessary for every 1 Gb/sec network throughput [53]. Experiments

[10]http://www.infinibandta.org/content/pages.php?pg=technology_overview

[11]Scheme that results in a 20 percent ((10-8)/10) overhead on the raw bit rate.

[12]Fall of 2011.

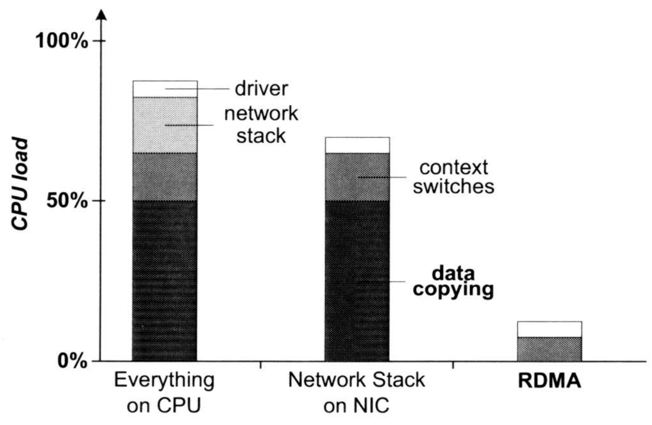


Figure 2.4: Only RDMA is able to significantly reduce the local I/O overhead induced at high speed data transfers[55].

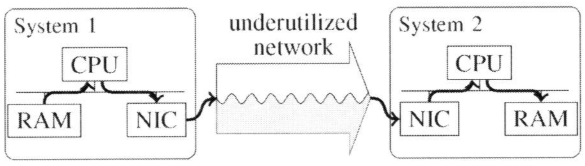
on our cluster confirmed this rule: even under full CPU load, our 2.33 GHz quad-core system was barely able to saturate the 10 Gb/sec link.

As can be seen in Figure 2.4, data movement causes roughly 50 % of the total CPU cost. In addition, the enforcement of process isolation mechanisms by the operating system kernel usually even requires three or more crossings of the memory bus. It can cause significant *bus contention* (assuming three bus crossings, 10 Gb/sec full-duplex communication leads to ≈ 7.5 GB/sec of bus traffic.) [55].

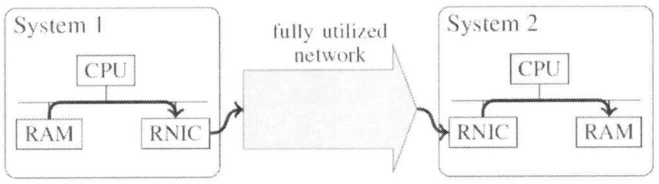
2.6.2 RDMA benefits

The most apparent benefit of using RDMA is the CPU load reduction thanks to the aforementioned direct data placement (avoid intermediate data copies) and OS bypassing techniques (reduced context switch rate) [13].

Figure 2.4 depicts the CPU load breakdown for legacy network interfaces under heavy load and contrasts them with the latest RDMA technology. As can be seen, the major CPU cost factor in traditional TCP handling is *not* network stack processing. The dominating cost is the intermediate data copying—required by most legacy transport protocols—to transfer data between the network and the local main memory. Therefore, offloading only the network stack processing to the NIC is not sufficient (middle chart), but data copying must be avoided as well. Only RDMA is currently able to deliver such a high throughput at negligible CPU load.



(a) Kernel TCP/IP. Data copying goes through CPU; several memory bus crossings.



(b) Network transfer using RDMA. RNICs handle data transfer autonomously; data has to cross each memory bus only once.

Figure 2.5: Network utilization [55].

A second effect is less obvious: RDMA also significantly reduces the memory bus load as the data is directly DMAed to/from its location in main-memory. Therefore, the data crosses the memory bus only once per transfer. The kernel TCP/IP stack on the other hand requires several such crossings. As indicated in Figure 2.5a, the data to be transferred crosses the system's memory bus at least two times on its way from the host memory to the network card (the same happens symmetrically on the receiver side). This may lead to noticeable *contention* on the memory bus under high network I/O. Thus, adding additional CPU cores to the system is *not* a replacement for RDMA.

2.6.3 Applying RDMA

Remote Direct Memory Access (RDMA) moves data from the memory of one host to the memory of another host by a specialized network interface card called *RDMA-enabled NIC (RNIC)*.

Before starting network transfers, application memory regions must be explicitly registered with the network adapter. This serves two purposes: first, the memory is pinned and prevented from being swapped out to disk; secondly, the network adapter stores the physical address corresponding to the application virtual address. Now it is able to directly access local memory using its DMA engine and to do remote data exchanges. Inbound/outbound data can directly be placed/fetched by the adapter to/from the address in main memory supplied by the application.

Figure 2.5b illustrates a typical RDMA data path: thanks to the placement information, the RNIC of the sending host can fetch the data directly out of local main-memory using DMA. It then transmits the data across the network to the remote host where a receiving RNIC places the data straight in its destination memory location. On both hosts, the CPUs only need to perform control functionality, if at all. Data transfer is done entirely by the RNICs. They can handle high-speed network I/O (≥ 40 Gb/sec) between two hosts with minimal involvement of either CPU.

2.6.4 Perfect match

By design, the RDMA interface is quite different from a classical Socket interface. A key difference is the asynchronous execution of the data transfer operations which allows overlapping of communication and computation, thereby hiding the network delay.

Taking full advantage of RDMA is not trivial as it has hidden costs [57] with regard to its explicit buffer management. The ideal scenario is when the buffer elements are large, preferably a few dozen up to hundreds of megabytes. Furthermore, to alleviate expensive memory registration, the node interconnections topology should be point to point [55].

Due to these costs and functional requirements, not every application can fully benefit from RDMA. However, an architecture like the *Data Cyclotron* clearly can. It aims to transfer big data fragments using a ring topology, i.e., each node has a point to point connection with its neighbors. It is a peer-2-peer style and not master server protocol on a 2D mesh [13] which is for example used by MapReduce frameworks. Hence, the Data Cyclotron is able to explore ideas from the peer-2-peer world, and thus be more suitable to cloud-style communication environments [56].

2.6.5 A burst for distributed processing

Through the years the data access latency for DBMS has been improved. Different algorithms or structures were created to exploit the hardware improvements for an efficient data access. RDMA, an advancement in network hardware, is a recent candidate to explore new methods for efficient data access.

The RDMA made possible to access to remote memory as if it was a local storage. Due to the DMA facility, data processing overlaps with data access. Furthermore, data

[13] M mappers send data to R reducers ($M \times R$ connections)

can be fetched to the local memory at higher speeds due to the high bandwidth provided by InfiniBand.

The Data Cyclotron [63] and *cyclo-join* [55, 54] are on the front line to capitalize those features. In the Data Cyclotron the data structures are placed into the remote memory without effort and the process is all transparent for the application. Multiple nodes handle large numbers of concurrent computations. Thanks to InfiniBand and RDMA, the Data Cyclotron realizes high speed and low latency data availability at all nodes.

In [55] for $(R \bowtie S)$, one relation, say S (partitioned into sub-relations S_i) is kept stationary during processing while the fragments of the other relation, say R, are rotating. Hence, all ring members join each fragment of R flowing by against their local piece of S (S_i) *locally* using a commodity in-memory join algorithm.

In the same line as RAMcloud [120], they exploit the fact that remote memory access is becoming more efficient than disk access, i.e., it has lower latency and higher bandwidth. Typical contemporary storage system architectures, like SATA300 and SAS, provide a theoretical maximum IO performance of 3000 Mb/sec (375 MB/sec). The effective bandwidth is further degraded by additional hardware overhead, e.g., the seek time, operating system software time, and especially the random access time.

Remote memory does not suffer from seek and rotational delays. Furthermore, with the sizable InfiniBand bandwidth (cf., Section 2.5) and the RDMA benefits (cf., Section 2.6.2), the latency for random data access to remote memories is lower than to the local disk [120]. They use the RDMA buffers, i.e., the registered memory regions, as a *drop-box* to shuffle data between nodes removing the dependency from slow disks. They explore the effortless and efficient data movement to develop new ideas for distributed data access and design of distributed relational operators.

2.7 Summary

The Chapter introduced related architectures, or concepts, developed in last decades for distributed data analysis. The most recent advances for scalability were brought to discussion for further analyze in Chapter 5. *DataPath* [11] and *Cooperative Scans* [155] were the two related concepts that exploit a continuous data stream to boost query performance. They stipulated a fact, to achieve high throughput it is required cooperative work among the nodes. On the other hand, Broadcast Disks [4] and DataCycle [75] claim the ring topology as the preferred one for a cooperative behavior in the distributed setting.

A well known MapReduce framework was dissected to understand the advantages and disadvantages of its architecture for certain workload scenarios. From data distribution to resource utilization, from data processing to integration with a DBMS, the major features were presented as well as the latest solutions to improve them. The analyze brought out the most important architectural design features of MapReduce-inspired solutions which do not make them suitable for complex data analyzes.

To complete the background reading, the trends in data centers at hardware and architecture level were summarized to make the dissertation self-contained. It was emphasized that network is becoming a relevant component to be taken in consideration when designing new distributed architectures. In the same line, the new application trends are motivating the exploration of new topologies and new techniques for data distribution. At long term, data centers will be equipped with diskless nodes and the switches in up layers will gradually be removed in favor of nodes with routing capacity. At the same time, fault tolerance will be supported by the hardware than the software. Combined with multipath technology, the data center will soon be seen as a cloud where peer-2-peer style environments can emerge as alternative solutions for distributed computation.

Chapter 3

The Data Cyclotron Architecture

This Chapter introduces the Data Cyclotron (DaCy) architecture which is built around a ring of homogeneous nodes. The Data Cyclotron has adopted ring topology for a decentralized architecture where nodes share data or parallelize data computations without a central coordinator. Its simplicity is the key to explore new and un-orthodox algorithms for distributed parallel processing. Its communication pattern leverages the data routing latency and gets optimal bandwidth utilization at networks switches using simplified routing.

For a flexible integration with different applications, each the Data Cyclotron node is defined by three layers, the network layer, DaCy layer, and application layer. Their interaction is achieved by a few support structures and routines having in mind an efficient platform for data load, data forward, and data access by all nodes. Together with the DaCy storage they provide the grounds to have a decentralized inter-node interaction to provide load balancing and scalability.

The Data Cyclotron aims both, to be a scalable architecture and it follows an adaptive vision to populate the nodes with data. With an independent service at the data source nodes to extract data fields from diverse data sets, a user can opt to load all data before defining the computations or load data as it is requested by the computations. The latter model reduces by several orders of magnitude the cumulative cost from the first data chunk load until its first utilization.

3.1 Outline

The remainder of this Chapter is organized as follows. Section 3.2 starts with the motivation to use ring topology as the logical topology and how such topology exploits the physical topology on state-of-the-art data centers. The Chapter continues with the description of the nodes internal organization in Section 3.3, followed by a description on how nodes interact in Section 3.4. For the two types of interaction, the DaCy storage is used as the communication hub. Its structural organization and its management is presented in Section 3.5. Finally, the Chapter concludes with the presentation of two models for data distribution, *a priory* loading and *iterative* loading in Section 3.6 and a summary in Section 3.7.

3.2 Logical topology

The relevance of communication protocols and logical topology in the computation process design has increased over the last few years. It has gained influence on how parallel computers are built for optimal balance on resource utilization, i.e., the amount of computation a node does relative to the amount data it can communicate with other nodes, also known as *grain size* [1].

From the communication perspective, in the ideal scenario, parallel computers should have a network diameter of two, i.e., every node would be connected to every other node. However, it would make large parallel computers impractical due to the exponential growth in the number of links.

To overcome the problem, developers started to explore programs which use communication patterns with the immediate neighbors, such as ring topology, to achieve an effective diameter of two. With the adoption of this communication pattern, parallel computers can approach their theoretical performance limit. Furthermore, they can exploit a *symbiotic* relationship between the communication process and the underlying hardware.

For these reasons, and to explore state-of-the-art network hardware, we have adopted the ring topology as the logical topology for the Data Cyclotron. The next section exposes the advantages of the topology and where its simplicity has been used to explore new research directions.

[1] A large grain size indicates the node does a lot of computation while communicating infrequently. A small grain size indicates that only a small amount of computation is done between communications.

3.2.1 Ring topology

The Data Cyclotron has adopted a ring topology for a decentralized architecture where nodes share data or parallelize data computations without a central coordinator. Its simplicity is the key to explore new and un-orthodox algorithms for distributed parallel processing. This simplicity is often left behind in favor of a more complex and highly dynamic topology such as $2D$ mesh. For highly dynamic topologies the distance between two nodes is O(1), i.e., low latency, however, the bus bandwidth quickly becomes a performance bottleneck. Topologies like $2D$ mesh are the preferred ones for programs exchanging thousands of small messages since the latency is the major cost. For the Data Cyclotron the nodes are intended to exchange big chunks of data. Thus, topologies that provide higher bandwidth between nodes, i.e., wider channels, and non-blocking traffic, are the preferable ones.

Symbiotic relationship between application and the network hardware.

For applications with communication patterns from one node to another node, the tree topology is the best strategy. It typically results in a route length of $O(logN)$ links with N being the number of nodes in the network. The ring topology is so suitable due to to its route length, i.e., $O(n)$ links.

The ring topology is more suitable for group (many-to-many) multicast communications. The overall number of links traversed by each packet is the same as in the tree topology, i.e., $O(N)$, where N is the size of the group. Furthermore, the bandwidth allocation on the ring for the multicast scenarios is $O(N)$, while on a general tree it is $O(N)$ for the static multicast scenario, but $O(N^2)$ for the dynamic and adaptive multicast scenarios [14].

The ring topology has another advantage, its communication pattern leverages the data routing latency and by using simplified routing it gets optimal bandwidth utilization at networks switches, i.e., the switches can operate in non-blocking mode. A network switch is non-blocking if the switching fabric can handle the theoretical maximum load on all ports such that any routing request to any free output port can be established successfully without interfering with other traffics. With non-blocking switches no extra latency is added by data routing at the switch.

Not all software architectures can use and exploit the assets of this type of switches, but the Data Cyclotron can. In the Data Cyclotron the data flows clockwise, i.e., it is a continuous stream and with a single routing pattern. Furthermore, with the ring topology packets that arrive at different input ports are destined to different output ports, i.e., a contention free scheduling. Therefore, they can be routed instantaneously, i.e, the switches can be non-blocking. Aside from absence of latency, there is ideal switch bandwidth utilization because the routing algorithm is equivalent to synchronized shuffling [38].

Simplified routing becomes more important for actual data centers due to its structural organization (cf., Section 2.4.2).

" A server is not anymore a single box with all compute resources in it. They have been placed in pools together with storage and memory processors. All components have been spread around the data center and they are all interconnected through a network. "

With the data center network trends (cf., Section 2.4) we believe the logical topology adopted by the Data Cyclotron is the best one to explore a new research path and guide us to a new optimal for distributed data analysis.

" Cloud computing platforms such as Windows Azure, Yahoo storage, or Google storage, have their storage services separated from compute nodes. To bring the storage network and compute server groups together the tendency is to use storage area networks (SANs). They use a fiber channel over Ethernet ($FCoE$) or "

Other architectures exploiting ring topology features.

The Data Cyclotron does not stand alone in the adoption of the ring topology to develop new and efficient distributed parallel processing algorithms. Recent multi-core processors are using a single ring to share data among their cores [140] such as Celli [122] and Nehalem-EX [41] processors. Intel is using four rings to interconnect its *Sandy Bridge* [42] processors. It uses four rings to split data movement from requests, acknowledgments, and snoops [140].

Its adoption for multi-core processors due to its simplicity and opportunity to explore novel designs raised some concerns about a possible higher latency compared to a more dynamic topology such as $2D$ mesh. A recent study [94] has shown that for on-chip networks the latency for a ring topology is in some cases lower than the latency achieved on a $2D$ mesh [94].

At first, latency in a ring seems to be a problem due to the high network diameter as the average hop count is proportional to N (number of nodes) while on the $2D$ mesh is proportional to $\sqrt[2]{N}$. However, due to the wider channel, i.e., high aggregated bandwidth, lower serialization, and lower per-hop latency, the overall latency can be lower without affecting the throughput. The results are summarized in Figure 3.1.

The design requirements, and bandwidth, facing on-chip networks are few orders of magnitude different from the ones in a data center network. Nevertheless, some ob-

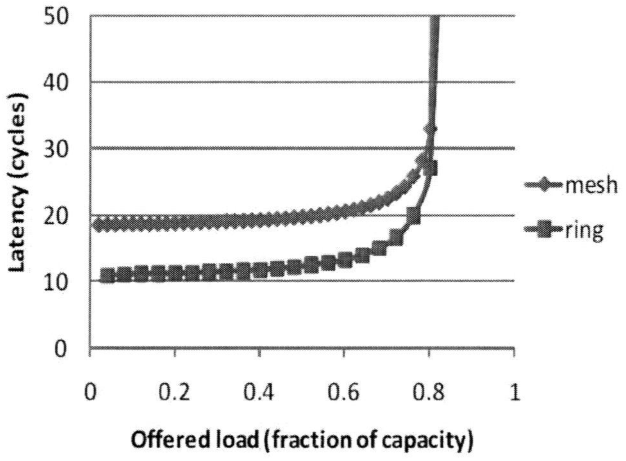


Figure 3.1: Latency vs. load curve comparing ring and mesh for a 16-node ring topology on uniform random traffic [94].

servations can be used in the Data Cyclotron context. The Data Cyclotron nodes, as the on-chip cores, load and transfer big data blocks at high speed to achieve high throughput and low latency. Moreover, the network technology used by the Data Cyclotron, i.e., RDMA, capitalizes the results through full link utilization and no serialization cost (more details in Section 2.6).

3.3 The Data Cyclotron node organization

To have a symbiotic relationship with network hardware and flexible integration with different applications, each Data Cyclotron node is defined by three layers: the network layer, DaCy layer, and application layer as represented in Figure 3.2. The network layer is responsible for all in and out node's traffic. It provides an API for the DaCy layer to control all network actions and status. The application layer uses three calls to communicate what data is needed, which data is in use, and which one was released. The DaCy layer is between these two layers and its runtime system, composed of several independent threads, controls the flow of messages.

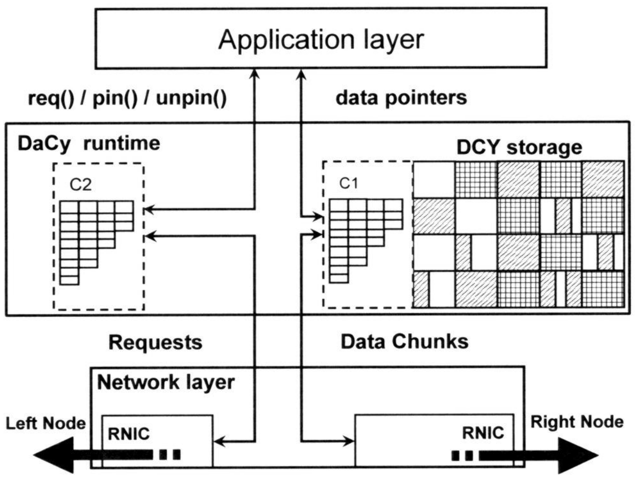


Figure 3.2: The Data Cyclotron architecture.

3.3.1 Network layer

The network layer is responsible for connections with the node's immediate neighbors. It encapsulates the envisioned RDMA infrastructure and traditional UDP/TCP functionality as a fall-back solution. The ring set up and the communication model derives from the network layer used for [55, 56] [2].

Ring set up.

During the ring initialization, with the exception for the first node, each new node connects to the last and first node added to the ring, as shown in Figure 3.3. Node $N3$ connects first to node $N2$ and then to node $N1$. Once $N1$ accepts $N3$ connection, it drops the connection with $N2$ (cf., steps C), D), and E) in Figure 3.3). The counter clockwise connection is initialized once the clockwise connection is set.

Each node before entering the ring allocates a set of memory regions to be used as DaCy buffers, and if the RDMA protocol is the chosen one, register them with the

[2]Result of a productive cooperation between CWI - Amsterdam and ETH - Zurich, in the initial stages of the Data Cyclotron project (http://www.systems.ethz.ch/research/researchareas/DPMH/DataCycl)

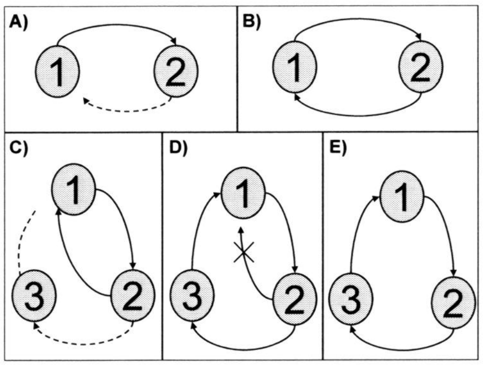


Figure 3.3: Ring extension.

network adapter [3] (cf., Section 2.6.3).

Communication model.

The communication model has been designed as an asynchronous scheme which involves two entities at each node: the *receive thread* (RX) and the *transmit thread* (TX) as picture in Figure 3.4. Both threads are independent from each other. RX receives a message, places it into a buffer, and notifies the *routine caller*. There is one routine caller for the counter-clockwise traffic and another one for the clockwise traffic. A *routine caller* inspects the message header and determines each routine from the DaCy layer should process it. Through flags the routines from DaCy layer know which DaCy buffers they need to process.

On the other hand, TX is responsible to forward messages to the next node. It inspects all buffers looking for the ones marked ready for forwarding. It interacts with the buffer manager for garbage collection or data caching.

Management of the network traffic.

The in/out traffic is composed of data chunk messages and data chunk request messages [4]. New type of messages can be added to the network traffic upon request. Their

[3]All nodes in a ring need to use same communication protocol

[4]For the remind a data chunk request will be referred as request.

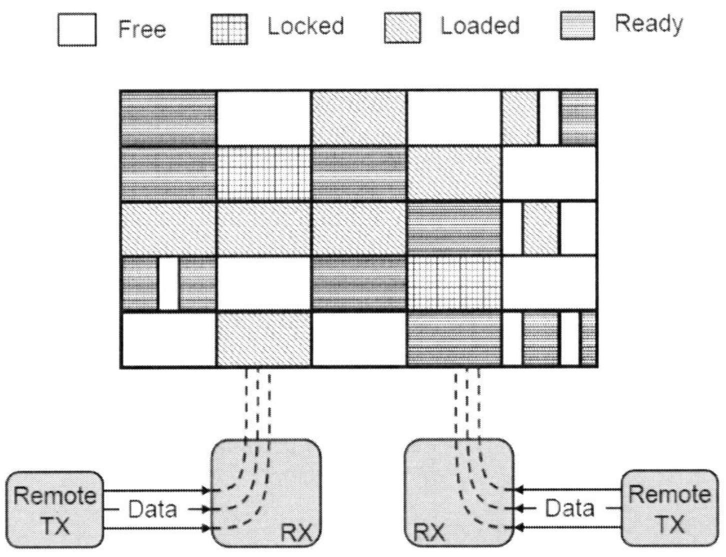


Figure 3.4: Asynchronous data propagation scheme.

flow is always counter clockwise to not increase the data propagation latency. For each new type, a routine is added to the DaCy runtime system and the *routine caller* made aware of the existence of a new type of message.

All messages are managed by the network layer on a first-come-first-serve basis. The exchange of messages is all transparent for the DaCy layer. The underlying network is configured as asynchronous channels with guaranteed order of arrival. The data transfer and the queues management are optimized depending on the protocol being used.

3.3.2 DaCy layer

The DaCy layer is an implicit interface for the application. It infers network sends/receives without the application knowledge. It is the control center and it serves three message streams, those composed of a) the requests from the local application instance, b) the predecessor's data chunks, and c) the successor's requests from the network layer.

To manage these streams it uses two catalogs [5] *C1* and *C2*. *C1* contains information about all data chunks owned by the local node. *C2* administers the outstanding requests for all active computations. They are organized by data chunk identifier.

The runtime system through the routines, updates each catalog based on the message's arrival. Furthermore, it is also responsible for the inter-node interaction and the storage ring management, both of them are discussed in detail in Section 3.4 and Section 3.5.

3.3.3 Application layer

For an application to interact with the Data Cyclotron it needs to be integrated with the application layer for a transparent interaction with all other layers. The application layer interacts with the DaCy layer through three calls *request()*, *pin()*, and *unpin()*.

A *request()* call is used to inform which data will be required. It is used by the DaCy layer to warm up local cache through a local or a remote data load. The *pin()* call is used to check the local cache for data availability. If it is not available, the application call blocks.

The application call remains blocked until the data chunk is made available at the DaCy buffers. Once available, the application call resumes and uses the data chunk for processing. The data chunk is only cached or dropped after the *unpin()* call. Until there the DaCy layer assumes the data chunk is being used for processing by the application. The interaction between the application layer and the DaCy layer ends with the last *unpin()* call.

3.4 Inter-node interaction

The inter-node interaction is done through two data flows, one counter clockwise for data requests, queries/jobs and metadata, and another one clockwise only for data chunks. Within these flows only two type of messages define the interaction, requests and data chunks. All other messages are application specific and they do not directly interfere in the inter-node interaction.

The interaction can be resumed by a single request for a data chunk. A request of a local data chunk, leads to an automatic load from disk into the local buffers, otherwise, the request leads to an update of the requests catalog. In the latter case, a data chunk request message is sent off to the ring, traveling counter-clockwise towards data chunk owner. Thereafter, the data chunk is loaded into the ring and travels clockwise towards the requesting node.

[5] One catalog per each type of message

Between the request load and the data chunk reception, all nodes interact through a *request propagation algorithm* and a *data chunk propagation algorithm* for efficient requests and data forwarding.

3.4.1 Request forwarding

The requests are used as informative pings by the application to keep the DaCy layer informed about which data chunks still need to be processed. These requests are stored in a catalog and used to identify which of the data chunks passing by needs to be retained in the local memory for processing.

After the request registration, the runtime system keeps track of the requests passing by. The routine *request propagation* (cf., Figure 3.5), analyzes the counter-clockwise traffic to see if similar requests are flowing. In case it sees an already loaded request and the data chunk has not yet arrived, it absorbs the remote request to reduce traffic in the counter-clockwise ring. Otherwise, the request is forwarded to notify the other nodes that a certain data chunk is required for data processing.

Hence, the requests flowing in the ring are aggregated to reduce the latency in requesting data. At the data chunk's owner, the request is removed from circulation and the runtime system attempts to load the data chunk. In case the data chunk was already loaded the request is just ignored, if not, and if the storage ring has space to transport it, the data chunk is sent to the loading list, otherwise, it is sent to the pending list.

To avoid data starvation, the runtime uses a time-out to reload a request. After TR timeouts the data chunk is declared as non-existent, the request un-registered from the catalog and an exception is raised: The data chunk does not exist anymore in any of the connected nodes.

3.4.2 Data forwarding

Once a data chunk is loaded, it travels from node to node until it is not needed anymore, thereby removed by its owner. The data chunk propagation from the predecessor node to the successor node is carried out by the *data chunk propagation* algorithm as depicted in Figure 3.6.

For each data chunk received, the algorithm searches for an outstanding request. Once found, it checks the catalog in search for computations blocked in a *pin()* call. For those computations, it unblocks them by handing over the pointer for the buffer where the data chunk is stored.

A data chunk carries an administrative header used by its owner for hot-set management. The header contains some properties, e.g., *data_chunk_id*, *data_chunk_size*,

Request Propagation

input: the request node's origin `owner` and the request id `reqid`
output: forwards the request or schedules the load of the requested BAT

```
/* check if the request returned to its origin */
if ( owner == node_id )
   unregister_request( &S2, reqid );
   unregister_request_queries ( &S3, reqid );
   exit;

/* check if the node is the BAT owner */
if ( node_is_owner_( &S1, reqid ) )
   if ( data_chunk_is_loaded( &S1, reqid) )
       exit;
   if ( data_chunk_can_be_loaded( reqid ) )
      if ( data_chunk_is_already_pending(reqid) )
         data_chunk_load( reqid, chk_size);
      untag_data_chunk_pending(reqid);
      exit ;
   else
       if ( !data_chunk_is_already_pending(reqid) )
          tag_data_chunk_pending(reqid);
          exit;

/* check if there is the same request locally */
if ( request_is_mine(reqid) )
   if ( !request_is_sent(reqid) )
      /* send if it has not been sent */
      load_request(node_id, reqid);
      exit;

forward_request(owner, reqid);
```

Figure 3.5: Request Propagation Algorithm

copies, *hops*, and *cycles*. The *data chunk propagation* algorithm (Fig. 3.6) updates the resource variables *hops* and *copies*. The variable *copies* designates how many nodes actually used the data chunk for their local computations. The variable *cycles* is update everytime it passes at its owner, i.e., a metric for the data chunk's age in the storage ring.

The DaCy runtime is aware of the memory consumption in the local node only. If there is not enough buffer space, the data chunk will continue its journey and computations waiting for it remain blocked for one more cycle.

Data Chunk Propagation
input: the BAT loader's id owner, data_chunk_id, loi, copies, hops ,cycles
output: forwards the BAT

```
/* check if there is a local request for the BAT */
hops++;
if ( data_chunk_has_request(data_chunk_id) )
   request_set_sent(data_chunk_id);

   if ( request_has_pin_calls( data_chunk_id ) )
      copies++;
      /*check if it was pinned for all the associated queries */
      if ( request_is_pinned_all( data_chunk_id ) )
         request_unregister( data_chunk_id);

forward_data_chunk(owner, data_chunk_id, loi, copies, hops, cycles);
```

Figure 3.6: Data Chunk Propagation Algorithm

3.4.3 Data loading

The runtime system has two more functions for resources management. A *resend()* function is triggered by a timeout on the rotational delay for data chunks requested into the storage ring. It indicates a package loss. The *loadAll()* executes postponed data chunks loads, i.e., data chunks marked as pending in the third outcome of the *request propagation* algorithm. It starts the load for the oldest ones. If a data chunk does not fit in the *data chunk queue*, it tries the next one and so on until it fills up the queue. The leftovers stay for the next call. This type of load optimizes the buffer utilization.

The priority for entering the storage ring is derived from both size and waiting time. This generic priority policy is defined for robustness, however, it might not be optimal for the data access latency. The data relevance for the workload should also be used as an additional weight in the priority setting. The boundaries of such optimization is presented together with the *hot-set management* in Chapter 4 to maintain the Chapter self-contained.

To reduce latency, it is assured that the load of a data chunk is not postponed indefinitely. In such situation, the *loadAll()* blocks until enough buffer space is released. These functions make the Data Cyclotron robust against request losses and starvation due to scheduling anomalies.

3.4.4 Ring's heartbeat

The definition of *time-outs* in a decentralized structure is always complex. Without global knowledge on the number of participants, and how their load is affecting the network traffic, using predefined *time-outs* can lead to data starvation or buffers and queues overflows.

The unit to define *time-outs* should be adjusted over the time. For that reason, a time unit called *DaCy-cycle* was introduced. A *DaCy-cycle* is the average time for a package to be sent and return back to the origin. Since requests do not necessarily complete a full cycle and data chunks can be unloaded, a special message is used for this case. With a time unit defined with a single message the *time-outs* among all node are aligned.

Hence, a special data chunk, called *DaCyPing* is loaded into the ring at the ring's initialization. Loaded by the first node that defines the ring. It travels with high priority and it is never removed from circulation. In the context of distributed systems, it can be seen as an *heartbeat* for the ring.

It contains an entry per node referring to network statistics at each of them. The number of entries is then used by the nodes as an estimation of how many nodes compose the ring as well as their load and bandwidth. Such information is relevant for adjustments on the ring structure (addition or removal of nodes) or in the flow of messages (number of messages or slowdown the forwarding). Its utility is emphasized in the coming Chapters.

3.5 Storage ring management

The storage ring is composed of all node's buffer space, i.e., all node's DaCy storage. The management of this space is done through the *buffer management* and the *hot-set management*. The buffer management assures an efficient utilization of the local buffers at each node. The hot-set management assures that only relevant data for the workload is flowing around the ring and its discussion has an entire dedicated Chapter 4.

3.5.1 Buffers organization

The DaCy storage is composed of several buffers and each of them has a list of properties. These properties are used to determine the used/free space and if they contain data *in transit* or data *in use* by the application. They are used by the DaCy buffer management to determine the right number of buffers for hot-set propagation and to feed properly the local starving computations. The global and local throughput depends on the success of this management decision.

The DaCy storage is divided into the *DaCy space* and *application space*. A low number of free buffers for *transit data* slows down the data propagation, therefore, it decreases the global throughput. On the other hand, few buffers to store data for the application degrades the performance of local computations.

The DaCy sets a minimal number of buffer space for each type of data. The remaining space, *neutral zone*, is used for both data types depending on the workload requirements. For a workload with a small hot-set the *neutral zone* is used to cache data chunks for the application. Used data chunks are kept to be re-used by future computations. If they are under node's ownership, they are ready to be forwarded up to request reducing data access latency. The cache management is explained in more detail in the coming Chapter 4.

On the other hand, if the hot-set grows the *neutral zone* starts to be used as a queue for data chunks in transit. Therefore, *application's data* is removed or forbidden to use the *neutral zone*. The first data chunks to be evicted are the *cached* data chunks followed by the ones *in use* by the application.

With the entire *neutral zone* occupied with transit data chunks, the storage ring is overloaded, therefore, new data chunks cannot be loaded. The Data Cyclotron reduces the number of data chunks in the ring by increasing a threshold, called $LOIT_n$, to unload the less popular data chunks from circulation. A detailed study of the dynamic adjustments of the threshold is presented in the coming Chapter 4.

3.5.2 Buffer fragmentation

The data chunks size is not equal and it is not known a priory. Furthermore, the buffer registration cost on the network adapter requires a buffer to be registered at the node's initialization. To cope with this issue and to reduce fragmentation, a node allocates the buffers with the same size and stores as many data chunks as possible in the same buffer.

Each buffer contains a *offset* list to delimit the occupied areas and the ones that remain empty. Every time a buffer is loaded the offset for the free space is updated. The management of these offsets is similar to the *buddy memory allocation* algorithm [144] and it is relatively easy to implement and supports limited but efficient splitting and coalescing of memory blocks [97].

The buddy memory allocation splits the buffers into halves, i.e., it creates *binary buddies*. Each buffer is divided into two smaller blocks, and each smaller block becomes a unique buddy to the other. A split block can only be merged with its unique buddy block to reform the larger buffer they were split from. A binary tree is used to locate used or unused memory blocks.

After the buffers allocation, and registration, a binary tree is created to represent all *buddies* blocks for each buffer. The depth of the tree (Dep), passed as configuration parameter, defines how many times a buffer is split in half. For efficiency the buffer size is power of two. Power-of-two block sizes make address computation simple.

In comparison to other simpler techniques, such as dynamic allocation, it has little external fragmentation. Furthermore, the merge of *buddy blocks* has little overhead since the maximal number of merges per buffer is $\log(Dep)$.

This organization and management of the empty regions is not optimal, but it has been robust for actual scenarios. However, for workloads where the data chunks do not have power off two sizes, the internal fragmentation cannot be avoided, specially when the data chunk is a little larger than a small block, but a lot smaller than a large block. More robust solutions to manage the empty/occupied buffers, such as the *malloc algorithm*, should be used to reduce, or even to evict, internal and external fragmentation. However, its implementation and study is not part of the content of this dissertation.

3.6 Data loading and distribution

The Data Cyclotron provides two data distribution models: *iterative loading* and *a priory loading*. The data distribution model to be used depends on the workload type. It is up to the user to decide if it is worth to load all data before defining the computations or use an iterative load, this is, load data as it is requested by the computations.

For workloads which request data from a data source which is always online and has a stable connection with one or more nodes in the ring, the iterative mode should be the preferred one. For both, a schema is first shared among all nodes. A request is defined based on this schema and might have as target different data sources. It is composed of the source file identification, parse expression, and split function. For example, a request to extract a relational column from a CSV file, or to extract records from a data block stored in a cloud storage file.

Independently of the data source, the Data Cyclotron provides a service to be installed at the data source called *data loader*. A *data loader* establishes multiple connections with a Data Cyclotron's ring and waits for data load requests. Using different readers, the *data loader* extracts the relevant data, such as properties or fields of CSV file, slices the extracted data into partitions, and send them as data chunks to the ring. Each data chunk is tagged with a $nodeID$ to make sure the distribution is uniform and each node has a disjoint set. During the first cycle, the node designated by $nodeID$ stores a copy in its disk and updates its catalog. From that point on, it is responsible to load/unload the data chunk into the *hot-set*. All other nodes only update their catalog. With a single scan of the data source, the data is distributed for future requests.

For *a priory loading*, a single data load request is issued by the user, or by a node at the reception of the first computation. Such request triggers the load of all data for the entire schema. For *iterative loading*, several data load requests can be issued by the nodes. Each request only requires the load of input data which was not yet loaded into the ring. For example, if two columns of a relational table are used for the first time, a request is sent to the *data loader* to only extract the two columns from the data source. The data access latency for the first data chunk request is higher than the consequent ones. However, the cumulative cost from the first data chunk load until its utilization is several orders magnitude smaller than the cumulative cost when using *a priory* loading. To keep the Chapter self contained, a more detailed discussion, in the context of relational databases, is presented in Chapter 6.

3.7 Summary

The Chapter introduced the Data Cyclotron architecture by posing arguments for the adoption of a ring as the preferred logical topology. With the nodes arranged in a ring structure each node's layer was described in detail. Their interaction is achieved by a few support structures and routines. Their importance and contribution for the layers interaction was presented having in mind an efficient platform for data loading, data forwarding, and easy access by all the nodes. Major attention was given to the DaCy buffers which compose the storage ring. Their management and structural organization is the key for a near optimal resource utilization and efficient data forwarding.

The population of nodes with data follows the adaptive vision of the architecture. The applications can request data from various sources and access it on an *iterative* or *a priory* loading mode. The iterative load is however dependent of the type of application and data sources. Further evolution of this topic is described in coming Chapters as well as the validation of all protocols through simulation and a full functional system for state-of-the-art data centers.

Despite recent developments in the research world to enforce new radical changes on the design of network topologies, the Data Cyclotron stands as one of few architectures without central coordination that has an elegant way to distribute and access data. The same time, it has the grounds of flexibility to explore new and novel data analyze techniques using state-of-the-art hardware.

Chapter 4

The Data Cyclotron Hot-set Management

In the Data Cyclotron to achieve high throughput and low latency the most relevant data for the current workload is identified and set with the highest priority to consume the available network bandwidth and storage ring space. Designated as hot-set, its decentralized management is responsible to assure fast access to the most relevant data and avoid data starvation for computations interested on data chunks with low relevance.

A ranking system is used to determine where a data chunk should be, i.e., in disk, memory, or storage ring (also in memory, but in circulation). Its self-organization in a distributed manner, keeping an optimal resource utilization, replaces gradually the data in the ring to accommodate the current workload.

The ranking system is composed of a ranking metric called the level of interest (LOI) and a threshold called the level of interest threshold ($LOIT$). LOI describes how popular was and is a data chunk for the current workload. Lower is its value more likely is the data chunk to be evicted. $LOIT$ is the borderline between the hot-set and cold-set. Each node keeps adjusting $LOIT$ to have as many data chunks as possible in circulation and low data access latency.

Using a well known network simulator (*NS-2*), we explore different approaches and methods to reach the most efficient and robust solution to rank the data chunks popularity and to define the optimal hot-set. From the integration of an innovative cache management to ownership request we assure a homogeneous hot-set composed of data chunks with the highest probability of utilization. The study is supported by simulation results for different workload scenarios.

4.1 Outline

The remainder of this Chapter is organized as follows. Section 4.2 presents the hot-set management algorithm, and LOI and $LOIT$ concepts in detail. Section 4.3 contains the study of hot-set management for different scenarios using a discrete event simulation. Furthermore, based on observations collected during simulation, we improved LOI definition. Subsequently, we present an intermediate state between hot- and cold-set called *warm-set* in Section 4.4. Section 4.5 presents an efficient cache management to avoid the relevant data forwarding be slowed down by the load and forwarding of unpopular data. Finally, the Chapter concludes with the presentation of a dynamic data ownership model to create an homogeneous hot-set in Section 4.6 and a summary in Section 4.7.

4.2 Level of interest

The data chunk is loaded once there is space in the storage ring and it flows as long its LOI is above its owner's $LOIT$. It becomes a candidate to be part of the hot-set as soon as a request for its load is received at its owner [1].

Its LOI is updated at each cycle completion, this is, every time it returns to its owner. The hot-set management determines the new LOI based on the previous LOI and global information collected during the data chunk's journey such as the number of $copies$, the number of $hops$, and the number of $cycles$. The variables $copies$ and $hops$ are updated at each node. The variable $cycles$ is only updated by the data chunk's owner when it completes a cycle. Hence, as described in [63] the LOI is calculated was follows:

$$\begin{aligned} \text{CAVG} &= \frac{copies}{hops} \\ newLOI &= \frac{LOI + CAVG \times cycles}{cycles} = \frac{LOI}{cycles} + CAVG \end{aligned} \tag{4.1}$$

The previous LOI for a data chunk represents the ring's interest on the data chunk during the previous cycles. However, the latest cycle has more weight than the older ones. This weight is imposed by multiplication of the number of copies average $CAVG$ in the last cycle by the actual $cycles$ value:

$$\frac{copies}{hops} \times cycles \tag{4.2}$$

[1]The data chunk's owner is the node responsible to load/unload it to/from the storage ring.

New level of interest

input: the data_chunk_id, loi, copies, hops, cycles
output: forwards the data chunk or unloads the data chunk.

```
/* Check if the node is the data chunk loader */
if ( node_is_the_loader(data_chunk_id) )
   cycles++;
   new_loi =
    (loi + ((copies / hops) * cycles)) /
    cycles;
   copies = 0;
   hops = 0;
   if ( new_loi < loit(n) )
      unload_data_chunk( data_chunk_id );
else
   forward_data_chunk(data_chunk_id, new_loi, copies, hops, cycles);
```

Figure 4.1: Hot-set Management

The division by the number of cycles was used to apply an age weight to the formula. The age weight decreases over the time to avoid monopolization of resources, this is, an old popular data chunk should give place to new popular data chunks. Hence, old data chunks carry a low level of interest unless renewed in each pass through the ring.

Data chunk removal.

Once the new LOI is determined, it is compared with $LOIT$. If lower, the data chunk is removed from circulation. If not, the LOI variable is set with the new LOI value and the data chunk is sent back to the ring. This *hot-set management* (represented by algorithm in Figure 4.1) is executed at the Data Cyclotron layer for all data chunks received from the predecessor node.

At each node, the runtime system derives the $LOIT$ value using as reference the local *DaCy storage* load. Its adjustment is dynamic and inversely proportional to the *DaCy storage* load. If it is too loaded, $LOIT$ increases. It is step wise increased until the pending local data chunks can start moving. If the data is being forwarded and loaded without delays it is decreased so the data is kept in rotation as long as possible.

A node keeps adjusting $LOIT$ to have as many data chunks as possible in circulation, but at the same time not more than the available DaCy buffers for transit data to keep the data access latency low. The precision of LOI to rank each data chunk and the dynamic $LOIT$ adjustment is analyzed and improved in the coming sections.

4.3 Simulation

The design of the Data Cyclotron layers interaction is based on a discrete event simulation. The core of this activity is a detailed hot-set study using NS-2 [2], a popular simulator in the scientific community. The simulator runs on a Linux computer equipped with an Intel Core2 Quad CPU at 2.40 GHz, 8 GB RAM and 1 TB disk. The simulator was used out of the box, i.e., without any changes to its kernel.

The simulation scenario has a narrow scope. It is primarily intended to demonstrate and stress-test the behavior of the Data Cyclotron internal policies. Moreover, higher scale experiments are not recommended for this type of simulators due to some internal overflows in the packet transmission and synchronization [3]. Further study for a bigger scope, using the conclusions from this Chapter, is conducted with a full functional Data Cyclotron on a real-size cluster (cf., Chapter 5).

The base topology in our simulation study is a ring composed of ten nodes. Each node is interconnected with its neighbor through a duplex-link with 10 Gb/sec bandwidth, $6\,\mu$ delay, and drop policy based on drop packets from the tail of the queue. Each node contains 200 MB for the *DaCy storage*, i.e., network buffers, which results in a total ring capacity of 2 GB. These network characteristics comply with a cluster from our national super-computer center [4], the target for the full functional system.

The analysis is based on a raw data-set of 8 GB composed of 1000 data chunks with sizes varying from 1 MB to 10 MB. They are uniformly distributed over all nodes, giving ownership responsibility over about 0.8 GB of data per node. The workload is restricted to computations that access remote data only since we are primarily interested in the adaptive behavior of the storage ring.

The first step in the experimentation is to validate correctness of the Data Cyclotron protocols using micro-benchmarks. We discuss three workload scenarios in detail. In the first scenario, we study the impact of the $LOIT$ on the computation latency and throughput in a ring with limited capacity. In the second scenario, we test the robustness of the Data Cyclotron against skewed workloads with hot-sets varying over time. In the third scenario, we demonstrate the Data Cyclotron behavior for non-uniform access patterns.

[2]NS-2 was developed by UC Berkeley and is maintained by USC; cf., http://www.isi.edu/nsnam/ns/

[3]From our experience a high number of packets per data chunk and the continuous flow between dozen of nodes for several cycles exposed overflows for the sequence number and *ACKs*

[4]http://www.sara.nl/

4.3.1 Limited ring capacity

The Data Cyclotron keeps the hot-data in rotation by adjusting the minimum level of interest for data chunks on the move. The level of interest threshold ($LOIT$) defines if a data chunk is considered hot or cold. A high $LOIT$ level means a short life time for them in the ring, and vice-versa. The right $LOIT$ level and its dynamic adaptation are the issues to be explored with this experiment.

The experiment consists of firing 80 computations per second on each of the 10 nodes over a period of 60 seconds, and let the system run until all 48000 computations have finished. We use a synthetic workload that consists of computations requesting between one and five randomly chosen remote data chunks. The net computation execution times, i.e., assuming all required data is available in the local memory, are arbitrarily determined by scoring each accessed data chunk with a randomly chosen processing time between 100 milliseconds and 200 milliseconds.

To analyze the impact of $LOIT$ on the Data Cyclotron performance behavior, we repeat the experiment 11 times, increasing $LOIT$ on all nodes from 0.1 to 1.1 [5] in steps of 0.1. Between two runs, the ring buffers are cleaned, i.e., all data is unloaded to the local disk.

Figure 4.2 shows the Data Cyclotron throughput for each $LOIT$ iteration, i.e., the cumulative number of computations finished over time. The line *registered computations* represents the cumulative number of computations fired to the ring over time.

The experiment shows that a low $LOIT$ leads to a higher number of pending computations in the system. For $LOIT = 0.1$ at instant 40 seconds, only 8000 out of the 30000 registered computations are finished. However, for $LOIT = 1.1$ at the same instant, almost 25000 computations were finished. We observe that the computation throughput is monotonously increasing with the increase of $LOIT$.

Data access latency is also affected by low $LOIT$ values. The graph in Figure 4.3 shows the computation life time distribution (histogram) for three $LOIT$ levels. The computation life time is its gross execution time, i.e., the time spent from its arrival in the system until it has finished.

The results show that a high $LOIT$ leads to lower life time of a computation. For example, the $LOIT = 0.1$ has a peak in the number of computations resolved in less than 5 seconds, but then it has the remaining computations pending for at least 100 seconds.

The reason for these differences stems from the amount of data removed from the ring over the time and the data chunks size. The workload hot-set is bigger than the ring capacity which increases the competition for free space in the ring. Using a low $LOIT$, the removal of the hot data chunks is delayed, i.e., the pending loads list at each

[5] This is the upper limit of the function.

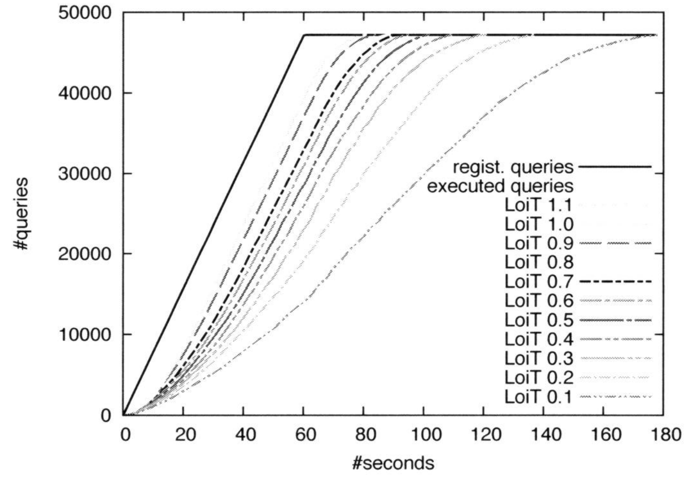


Figure 4.2: Query throughput for multiple $LOIT$ levels.

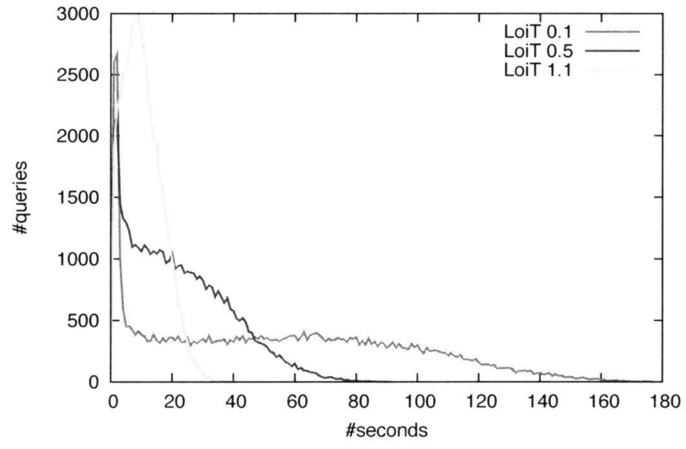


Figure 4.3: Query life time for multiple $LOIT$ levels.

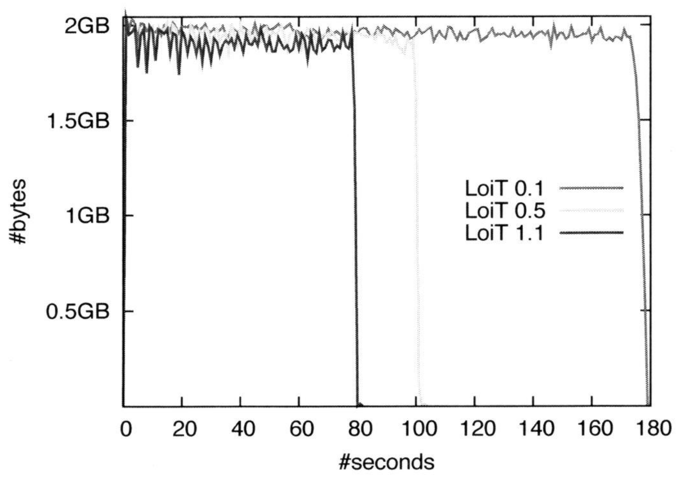


Figure 4.4: Ring Load in Bytes.

node grows. Consequently, execution of computations that wait for the pending loads is delayed.

With the optimized loading process for pending data chunks and a low drop rate, the tendency is to leave the big ones for last (cf., Section 3.4.3).

" The loadAll() *executes postponed data chunks loads, i.e., data chunks marked as pending in the third outcome of the* request propagation *algorithm. Every TC milliseconds, it starts the load for the oldest ones. If a data chunk does not fit in the* data chunk queue, *it tries the next one and so on until it fills up the queue. The leftovers stay for the next call. This type of load optimizes the buffer utilization. "*

Whenever the least interesting data chunk is dropped from the ring, the available slot can only be filled with a pending data chunk of at most the size of the dropped one. Consequently, the ring gets loaded with more and more small data chunks, decreasing the chance of loading big data chunks even further. Only once there are no more pending requests for small data chunks the ring slowly empties making room for the big data chunks waiting to be loaded.

The graphs in Figure 4.4 and Figure 4.5 identify the data chunk size trend over time. The correlation between the ring load in bytes (Figure 4.4) and the ring load in

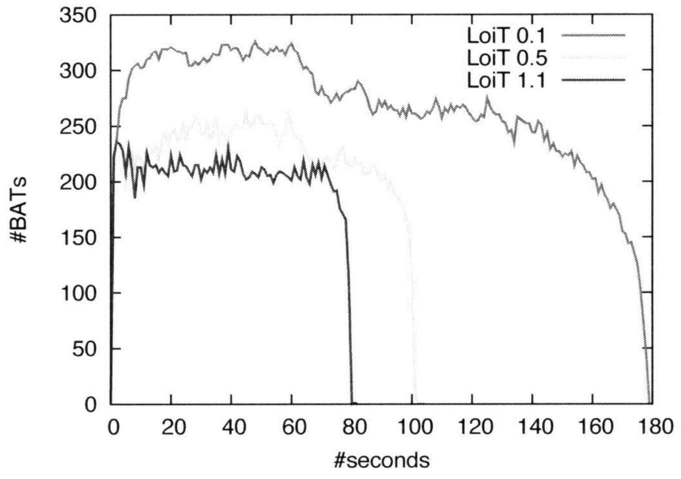


Figure 4.5: Ring load in chunks.

data chunks (Figure. 4.5) shows the data chunk length in the hot-set over time. With a continuously overloaded ring and a reduction of the number of data chunks loaded, the graphs depict that the load of big data chunks is being postponed. Therefore, the computations waiting for these data chunks stay pending almost until the end. The delay gets more evident for low $LOIT$ levels. It shows that a priority police defined only on size is not robust enough to reduce the data access latency. Instead, the priority policy should be based on their size and age.

The use of such a priority policy reduces the latency, but it does not improve the throughput in all situations. The reason is the absence of knowledge on how relevant the data chunk is for the throughput. Hence, the relevance of the data chunk for the hot-set should also be considered in the priority definition.

The experiment confirmed our intuition that the $LOIT$ should not be static. It should dynamically adapt using the local *DaCy storage* load as a reference. In the next experiment we show the $LOIT$ dynamic behavior when the hot-set is constantly changing and how well it exploits the ring resources.

workload	SW1	SW2	SW3	SW4
skewed	3	5	7	9
start(sec)	0	15	37.5	67.5
end(sec)	30	45	67.5	97.5
computations/sec	200	300	400	500

Table 4.1: Workload details.

4.3.2 Skewed workloads

The hot-set management is also tested for a volatile scenario. In this experiment several skewed workloads *SW* were used. A skewed workload *SWi* uses a subset of the entire database. The hot-set *Hi* used by *SWi* has disjoint data *DHi* not used by any other skewed workload. In addition, brute changes in the hot-set *H* and resource competition by disjoint hot-sets *DH* are also used.

Each workload *SWi* can enter the Data Cyclotron at a different time. In some cases they meet in the system, in other cases they initialize after the completion of the previous ones. This unpredictable initialization requires a dynamic and fast reaction by the Data Cyclotron. If *SWj* enters the ring while *SWi* is still in execution, the Data Cyclotron needs to arbitrate shared resources between the *DHi* and *DHj*. It must remove *DHi* data chunks with low LOI to make room for the new *DHj* data in order to maintain a high throughput. However, the data chunks from *DHi* to finish *SWi* computations must remain in the ring to ensure low query response time.

This dynamic and quick adaptation should provide answers to three major questions: How fast does the Data Cyclotron reacts to data requests for the new workload? Are the computations from the previous workload delayed? How does the Data Cyclotron exploits the available resources?

The scenario created has four workloads (*SW1, SW2, SW3, and SW4*). Each *SWi* uniformly accesses a subset of the database (*Di*). Each *Di* has a disjoint subset *DHi*, i.e., *DHi* is not in *Dj, Dk, Dl*, with the exception for *DH4* which is contained in *DH1*. Each *Di* is composed of data chunks for which the modulo of their *id* and a *skewed* value is equal to zero. The time overlap percentage between the *SW1* and *SW2* is 50%, 25% for *SW2* and *SW3*, and no overlap for *SW3* and *SW4*. Table 4.1 describes each workload.

From the previous experiment, we learned that the $LOIT$ should be inversely proportional to the buffer load. The dynamic adaption of the $LOIT$ is done using the local buffer load at each node. Every time the buffer load is above 80% of its capacity, the $LOIT$ is increased one level. On the other hand, if it drops below 40% of its capacity, the $LOIT$ is decreased one level. For this experiment, we used three levels, *0.1, 0.6, 1.1* and each node independently adjusts its $LOIT$. The reason to only use three levels is to have the node's $LOIT$ aligned, this is, nodes will have similar adjustments to $LOIT$, and thus have better isolation to analyze the behavior of the ring as a whole.

Graph 4.6 shows the space used in the ring by each *DHi*. While, Graph 4.7 shows the amount of computations finished for each *DHi*.

The results illustrate the reactive behavior, i.e. how quickly the Data Cyclotron reacts to changes in the workload characteristics. The graph in Figure 4.7 shows a peak of 2000 finished *DH2* computations between the 15th and 16th seconds. The graph 4.6 shows a peak in the load of *DH2* data chunks in the same period. With the initialization of *SW2* at the 15th second, the peak confirms the quick reaction time. The same phenomenon is visible for all other workloads.

The ring was loaded with data from *DH2*, however, the data from *DH1* was not completely removed. It is a consequence of the 50% time overlapping between *SW1* and *SW2*. In Graph 4.7 *SW1* computations remain visible until the 43rd second. The data chunks to resolve these computations are kept around as it is shown in Graph 4.6. The Data Cyclotron does not remove all data from the previous workload in the presence of a new workload until all computations are finished. It shares the resources between both workloads as predicted. Observe that the sharing of ring resources gets lower as the time overlapping between *SW* decreases.

The *SW3* workload shows an interesting reaction of the Data Cyclotron when it encounters a nearly empty ring. The *DH3* started to be loaded and the ring is near to its limit. In all nodes the $LOIT$ is at its maximum level to free space as much as possible. No more *SW1* and *SW2* computations exist in the system. Therefore, the last data chunks for *DH1* and *DH2* start to be removed from the ring. Their removal brings the ring load down to 37,5% of its capacity, below the 40% barrier defined for this experiment. With this load each node starts to set its $LOIT$ to its minimum level, i.e, the data chunks are now staying longer in the ring.

With a big percentage of *DH3* computations finished, the Data Cyclotron does not remove the *DH3* data chunks anymore. It keeps loading the missing *DH3* data. The ring gets loaded and it remains with the same load for almost 10 seconds. The Data Cyclotron exploits the available resources by maintaining the *DH3* data chunks longer, expecting they will be used in the near future.

The abundance of resources is over when the *SW4* workload enters the scene. The ring becomes overloaded again, raising the $LOIT$ to higher levels. Therefore, the *DH3* data starts being evicted.

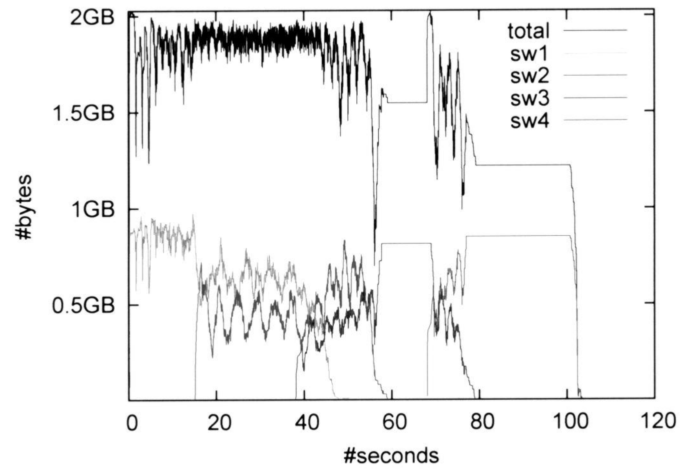


Figure 4.6: Ring load for skewed workload.

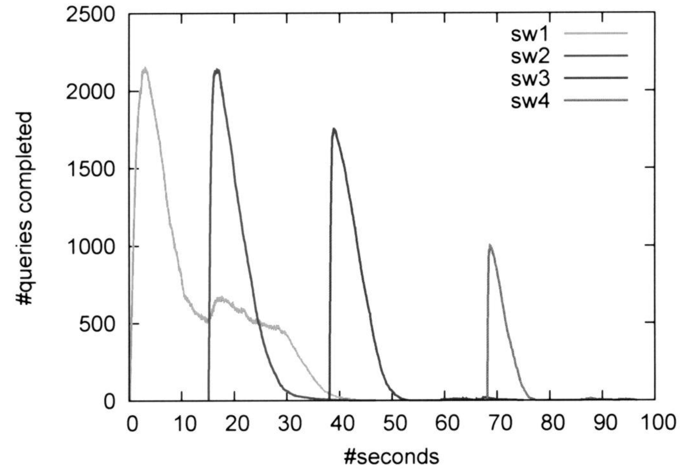


Figure 4.7: Computation throughput for skewed workload.

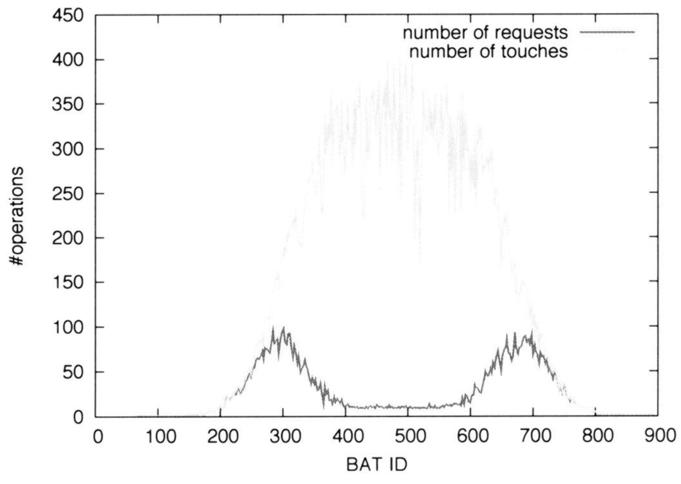


Figure 4.8: *Gaussian* workload: copies/requests distribution.

4.3.3 Non-uniform workloads

So far we have studied the hot-set management using uniform distributions for the data chunk size and data access patterns. Leaving the uniform scenarios behind, we move towards workloads with different data access distributions.

In the previous experiments, the study of the data chunk *LOI* focused on their age. The average number of copies per cycle, i.e., the ring interest within a cycle, was not included due to the uniform data chunk access pattern. Therefore, we initiated an experiment to stress it using a *Gaussian* data access distribution. The uniform distribution for data chunk size is retained. The workload scenario of Section 4.3.1 is used with the new data access distribution. The *Gaussian* distribution is centered around data chunk id 500 with a *standard deviation* of 50. All nodes use the same access distribution.

In this kind of workload, the Data Cyclotron keeps popular data chunks longer in the ring and exploits the remaining ring space for less popular data chunks. The workload distribution is represented by the gray curve in graph 4.8. The *in vogue* group is constituted by the data chunks with id between 350 and 600 which were copied more than 250 times. The data chunks on the edge of this group, the *standard* data chunks, have a lower rate of copies. The remaining ones, with less than 20 copies, are the *unpopular* data chunks.

The *in vogue* data chunks are heavily used by the computations, i.e., their *LOI* is

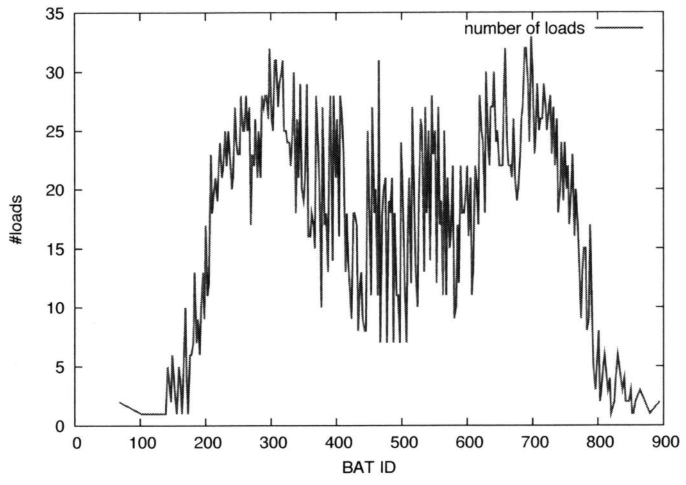


Figure 4.9: *Gaussian* workload: loads distribution.

always at high levels. Therefore, they are kept longer in the ring. Their low load rate, pictured as black in graph 4.9, is explained by the Data Cyclotron cold down process. With an overloaded ring and the $LOIT$ at its highest level, the Data Cyclotron removes data chunks to make room for new data. The first ones to be removed are the ones with low LOI, i.e., first the *unpopular* then the *standard* data chunks. For this reason the *in vogue* are the ones staying for longer periods as hot data chunks.

The *standard* data chunks are then requested by computations triggering their load. It is this resource management to maintain the latency in low values that makes the *standard* data chunks to enter and leave the ring more frequently.

The low rate of requests, represented as black, for the *in vogue* data chunks contradicts the common believe that *in vogue* data chunks should be the ones with the highest rate of requests, thereby a high rate of loads. The reason stems from the request management in the Data Cyclotron runtime layer. A request is only removed if all of its computations have *pinned* it. Having a high number of computations entering the system, the probability for an *in vogue* request to be *pinned* for all interested computations is too low. As a consequence, the request stays longer in the node's catalog and its load postponed because the data chunk is most of the time seen within a DaCy-cycle.

The experiment results show a good hot-set management for a *Gaussian* distribution. The high throughput is assured by keeping the *in vogue* data chunks in the ring as long as possible. For a low latency and large number of *standard* data chunks, the

$LOIT$ is used at its high level to reduce the access time to them. The *in vogue* data remains in the hot-set despite the high level of $LOIT$.

4.3.4 New level of interest

For a long number of DaCy-cycles, it was observed that an *in vogue* data chunk is exposed to the risk of being unloaded due to a drastic deviation on the ring interest on its latest cycle. In the presence of an outlier cycle, this is, sporadically the ring interest in the chunk drops or increases drastically, the hot-set management uses the data chunk's history to absorb a drastic deviation. However, the weight of the data chunk's history decreases over the time due to its division by the number of cycles (cf., Section 4.2).

$$newLOI = \frac{\mathit{LOI}}{\mathit{cycles}} + \mathit{CAVG} \tag{4.3}$$

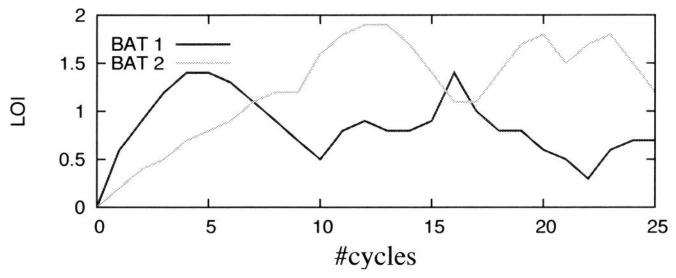
To improve the absorption of drastic deviations LOI was modified to use the relative change between the number of copies between the actual cycle and the previous cycle. Furthermore, the history weight is not anymore directly dependent of data chunk's age in the hot-set. Its weight is now controlled by a constant K. The modification does not change the general behavior of the hot-set management observed in previous experiments. It only makes LOI calculation more robust against drastic deviations after a high number of DaCy-cycles.

The new formula uses data chunk properties; number of *copies* from the actual cycle (C_x) and previous cycle (C_{x-1}), number of *hops* (H), and the previous LOI. The number of cycles is not used anymore. The $newLOI$ is then calculated as follows:

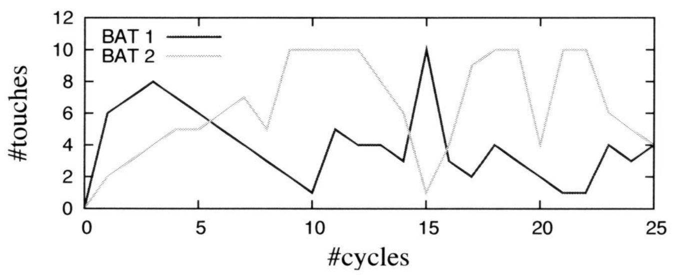
$$\begin{aligned} \text{RELCHG} &= \frac{C_x}{C_{x-1}} \\ \text{CAVG} &= \frac{C_{x-1} - (1 - RELCHG)}{\mathit{H}} \\ newLOI(x) &= \frac{\mathit{LOI}}{\mathit{K}} + \mathit{CAVG} \end{aligned} \tag{4.4}$$

We first determine the relative change, $RELCHG$, between the actual number of *copies* C_x and the previous number of *copies* C_{x-1}. It normalizes C_x with C_{x-1} to avoid the cases where the absolute difference is too big. If the number of copies increased relative to the previous cycle, C_{x-1} is increased by $RELCHG$, otherwise, decreased:

$$C_{x-1} - (1 - RELCHG) \tag{4.5}$$



(a) LOI per cycle



(b) Copies per cycle

Figure 4.10: LOI variation.

The previous LOI for a data chunk carries its ring's interest history during previous cycles. The variable K can be adjusted or modified at configuration time. It can be used to tune hot-set management to be more aggressive in using history to attenuate deviations. In the simulation we have used $K = 2$, this is, $newLOI$ contains half of the previous LOI to propagate the ring interest history over the time.

A simple simulation of two data chunks flowing in a ring can show how the formula introduces a smooth fluctuation on their $LOIs$ for drastic deviations on the ring interest from cycle to cycle, i.e., the number of copies per cycle. Figure 4.10 pictures the correlation between their LOI and the number of copies per cycle.

For the first ten cycles, the LOI of data chunk one ($CHK1$) increased almost linearly despite the variations on the number of copies between cycle six and cycle ten. A clear absorption of drastic changes in the data chunk popularity is seen between the cycle fourteen and cycle twenty. The LOI, for both data chunks had a light deviation despite the data chunks popularity inversion. Furthermore, it also shows how the data chunk's history has a weight in the new LOI determination. Between cycle fourteen and nineteen, despite the drastic changes in the data chunk's popularity, data chunk two ($CHK2$) only for a cycle had a bit higher LOI than $CHK1$.

Controlled by a constant, the data chunk history works as an optimistic adviser for the workload requests in the coming cycles. Data chunks that were popular in the last cycles might also be popular in the coming cycles and vice-versa. Hence, for situation like the one observed at cycle fifteen, a data chunk is not removed due to a momentarily tumble on its popularity.

K's value is a configuration parameter. Higher is its value more sensitive becomes the hot-set to drastic deviations on the workload. During the experiment, it was visible that for $K = 2$ LOI had fast reaction to workload peaks and good absorption of outlier cycles. The fast reaction is important for workload scenarios similar to the ones studied in Section 4.3.2.

4.3.5 Conclusion

The three scenarios have shown how the $LOIT$ needs to be adjusted and how it influences the throughput for skewed and non-uniform workloads. LOI was improved for a better control on the history propagation. Its new definition expanded capabilities of the hot-set management to be flexible for a bigger scope of workload scenarios. The model presented to manage the hot-set might not be optimal, but it is robust and behaves as desired. It is an open research challenge to find the optimal hot-set management. In the coming sections we propose further improvements and discuss how they are integrated to improve throughput and reduce response time.

4.4 Warm data

The Data Cyclotron at load time assumes that all data chunks to be loaded have the same probability to become *standard* or even *in vogue* data chunks. However, not all data chunks will then be classified as such. In case the loaded data chunk is an *unpopular* one low data access latency is assured for few computations, but overall it downgrades throughput. To overcome this issue we propose a pre-warming up phase before the data chunk load into the hot-set. In this phase, the data is in the *warm* state, an intermediate state between *cold* and *hot*, i.e., the data is in memory, but not in circulation.

A request, during its journey, collects an estimation of the interest on the data chunk. Once it reaches the data chunk owner, it triggers the load of the data chunk from the *cold-set* to the *warm-set*. The estimation is then used to pre-calculate a possible LOI as if the chunk had been loaded. For each DaCy-cycle its LOI is updated. Once it reaches a value higher than the ones flowing in the ring, the data chunk is loaded to the *hot-set*, replacing a data chunk with lower LOI. In case of equality, it is given priority to the new data chunk.

The LOI calculation is thus a continuous process since the reception of the first request. When the data chunk is unloaded, the data chunk is not moved to the *cold-set*, but to the *warm-set*. Hence, the calculation continues with the reception of more requests. If hot again, the data chunk is sent back to circulation. The process ends once the probability to be used reaches zero, i.e., the data chunk becomes part of the *cold-set*.

In case of a low number of data requests to trigger the data chunk load, a time-out, defined as number of DaCy-cycles, forces the load to avoid data starvation. As it will be described in the coming Section 4.6, the load of this type of data chunks combined with dynamic ownership model has a low impact on the flow of data chunks with higher popularity.

4.5 Cache

A computation can be composed of a single instruction or a set of instructions which are executed in any order such as map phase in MapReduce. In this type of computations an instruction execution is only dependent on input data from the storage ring.

Nevertheless, it can also be composed of a set of instructions for which the execution is also dependent on other instructions output such as a query execution plan for the relational model. Two different computations requesting the same data chunk might issue the *pin()* calls in a different order due to inter-operator dependencies. In most of the cases, the last computation to issue the *pin()* call for a data chunk CHK might have just missed the opportunity to retrieve it from the DaCy buffers without requesting it. Therefore, the computation remains blocked until the data chunk passes by again or, in worst case scenario, be reloaded. In case of an *unpopular* data chunk, the reload will take storage ring resources from the popular ones.

As a result, the response time increases due to a tiny time difference between the data chunk's forwarding time and the *pin()* execution time. Runtime caching emerges as a solution for this type of applications to reduce data access latency. However, caching the latest used ones is sufficient due to the high competition for resources. Hence, two caching policies, efficient for the Data Cyclotron context, are here presented. *Beforehand cache* for generic workloads and a cache policy for non-uniform workloads.

Beforehand cache policy.

The data chunks passing by are cached in case they have been requested but not yet pinned. However, not all of them can be cached due to space limitations. Hence, the ones with a *pin()* call being issued in the near future have high priority to be cached.

The runtime system through the registration time of the *request()* calls, and on previous *pin()* calls, estimates when the data chunk will be used [6]. Based on the estimated time, the runtime system predicts if a data chunk will be used before it completes another *DaCy-cycle*. If not, and in the presence of enough resources, the data chunk is cached. The data access time is then resumed to a read from the local memory.

Policy for non-uniform workloads.

For large hot-sets and non-uniform workload scenarios as the one used in Section 4.3.3 caching the ones which will be used in the near future is not enough to improve response time. During experiments on Section 4.3.3, it was observed that the life time of *unpopular* data was too short compared to *standard* data chunks and even more compared to *in vogue* data chunks. With the introduction of the warm-set in Section 4.4, the time to re-load them is longer than the time to re-load *standard* data chunk. Hence, the *beforehand* cache policy by itself is not enough to reduce response time.

The authors in Broadcast Disks [4] had a similar observation. They observed the data access latency is reduced if the less popular pages are cached instead of the popular ones. For their pure pushed-based system, instead of using a standard page replacement policy which tries to replace the cache-resident page with the lowest probability of access, they propose a replacement strategy that replaces the cache-resident page having the lowest ratio between its probability of access (P) and its frequency of broadcast (X). This ratio is referred as PIX.

The PIX policy was demonstrated to be optimal under certain assumptions. However, it was not a practical policy to implement. It requires a perfect knowledge of the access probabilities and it has an expensive comparison to determine which pages should be evicted. Therefore, they designed and implemented LIX, an approximation of PIX.

In the Data Cyclotron, a pull-based system, there are two factors that make the implementation of PIX policy feasible. The first factor is LOI which shows the access probability. The second factor is the comparison granularity. In the Data Cyclotron we use data chunks of multiple Megabytes rather than Kilobytes pages. Hence, the deter-

[6]The estimation is more accurate if provided by the computation which is possible at registration time.

mination of which data chunks should be evicted is trivial. By caching the less popular data chunks, the Data Cyclotron reduces the data access latency and the number of requests in circulation. At the same time, the *in vogue* data is not slowed down by the load and forwarding of *unpopular* data.

Cache management.

The cached data chunks are kept in the neutral zone of the DaCy storage (cf., Section 3.5.1).

" The DaCy sets a minimal number of buffer space for each type of data. The remaining space, neutral zone, *is used for both data types depending on the workload requirements. For a workload with a small hot-set the* neutral zone *is used to cache data chunks for the application. Used data chunks are kept to be re-used by future computations. If they are under node's ownership, they are ready to be forwarded up to request reducing data access latency. "*

The lack of space in this zone requires the drop of data chunks to free more space. The first data chunks to be evicted are the ones with the lowest PIX followed by the ones with highest time to be executed.

The runtime cache is the first cache layer of the Data Cyclotron. The Data Cyclotron has another two layers, the hot-set and the warm-set. The organization of the cache layers in this three levels reduces the data access latency for pull-based systems which exploit a continuous stream of data. The trade offs of these policies and their evaluation are part of on going research, part of the study is presented with a full functional system in the coming Chapter 5.

4.6 Homogeneous hot-set

In all scenarios until now, each node contributed with more or less the same amount of data. Hence, the $LOIT$ had the tendency to be similar among all nodes. Some workloads might have a narrow scope of interest and mostly request data from a subgroup of nodes, to be called *groovy* nodes.

With very dissimilar DaCy storage loads the definition of $LOIT$ becomes imprecise. All nodes have the same amount of data passing by, but a different load rate. The ones with high load rate tend to take $LOIT$ to higher levels than the others. The step wise adjustment triggered by the pending loads is not sufficient to have only the most relevant data in the storage ring (cf., Section 4.2).

" At each node, the runtime system derives the $LOIT$ value using as reference the local DaCy storage *load. Its adjustment is dynamic and inversely proportional to the* DaCy storage *load. If it is too loaded, $LOIT$ increases. It is step wise increased until the pending local data chunks can start moving. If the data is being forwarded and loaded without delays it is decreased so the data is kept in rotation as long as possible. "*

The $LOIT$ value should take as reference the minimum LOI observed in the hot-set. In case of a big discrepancy, such as three levels, it should be adjusted so the hot-set becomes more homogeneous. An homogeneous hot-set is composed only by the most relevant data. Like this, *non-groovy* nodes adjust their $LOIT$ and remove *unpopular* data chunks independently if their local buffers are overloaded or not.

For the improved version of $LOITn$ a node, per each DaCy-cycle, collects the LOI value of each data chunk passing by to determine the most frequently LOI value in the hot-set, i.e., the mode. The mode is used instead of the median, or mean, because the LOI distribution tends to be highly skewed. Its value and its standard deviation are used to adjust the $LOITn$. Since the mode value is not necessarily unique, and to maintain the node's autonomous behavior, the $LOITn$ is kept within an interval instead of being set with mode's value.

Figure 4.11 has a LOI's distribution example. Region $A)$ denotes unpopular data chunks while region $B)$ denotes popular ones. The *in vogue* ones are denoted by region $C)$. Region $B)$ is bounded by the standard deviation of mode. Hence, any node with the $LOITn$ outside region $B)$ is adjusted to be in the center of $B)$.

For a *non-groovy* node the $LOITn$ once set in the middle of region $B)$ remains there due to the lack of tension on the buffers to load data. On the other hand, for a *groovy* node the $LOITn$ has the tendency to be near the border between region $B)$ and $C)$ due to the tension on its buffers.

Request data ownership to load *in vogue* data chunks.

For overloaded rings and huge data sets the *groovy nodes* are under stress to load more data. When loading *in vogue* data, they have to wait for *unpopular* data chunks to complete their cycle and be unloaded by their owners. Hence, this waiting time is added to access latency of *in vogue* data. The situation becomes worst for large rings. To overcome the waiting time, they could sacrifice some of their *standard* data chunks, however, such decision would decrease throughput. Hence, the solution is to request the *unpopular* data ownership from remote nodes. Hence, a *groovy* node can unload a data chunk with a LOI significant lower than the ones it has to load. The data chunk

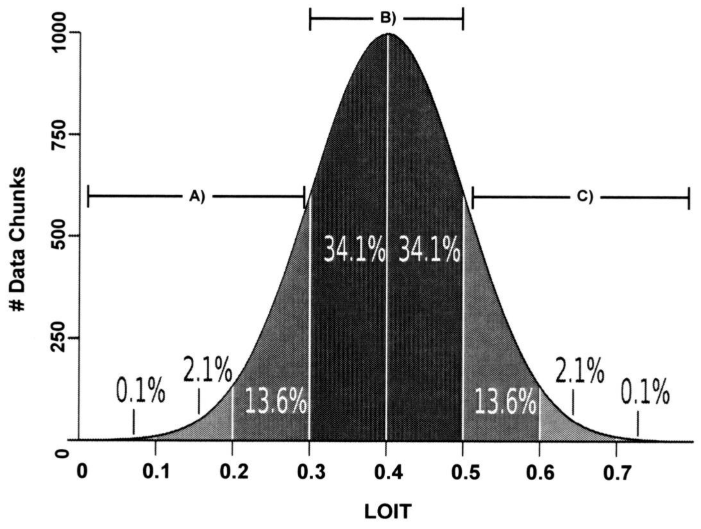


Figure 4.11: Standard Deviation for LOIT.

is thus moved to its warm set and the *groovy* node becomes responsible for its re-load into the *hot-set*. Mean while, the data chunk header continues its journey to inform its previous owner about the change of ownership.

A direct appliance of this ownership request is for data chunks loaded to avoid starvation (cf., Section 4.4).

" In case of low number of data requests to trigger the data chunk load, a time-out, defined as number of DaCy-cycles, forces the data chunk load to avoid data starvation. "

Their ownership is requested by remote nodes before completing a cycle to release space for the flow of relevant data. However, they cannot be unloaded before they have reached the nodes who have requested their load, or in an extreme case, the first one.

The model brings a new level of flexibility and robustness against rough workloads. Once the data is distributed at the ring initialization, the data can then bounce from one side of the ring to the other side and be owned by any node. The data chunk's ownership established at the data distribution time is not anymore lifelong.

At the same time, it is used to introduce the concept of speed lines, i.e., depending on the data chunk relevance, its cycle time can be longer than a DaCy-cycle. An *in vogue* data chunk will complete a cycle within a DaCy-cycle while a data chunk with lower relevance might need two DaCy-cycles. The difference of speed in each line is equal to the probability of a data chunk be moved down from the *hot-set* to a *warm-set* during its journey.

4.7 Summary

The Chapter presents the hot-set management using $LOIT$ as indicator how overloaded is the ring. The less popular chunks are identified through LOI and removed from circulation. In autonomous and dynamically way each node contributes with the best of its knowledge for an efficient definition of the hot-set.

A decentralized hot-set management can lead to an unfair management of less popular data when the storage ring is overloaded. To circumvent the problem a new state for the data was created as well as the integration of an innovative cache management. The cache management equips the Data Cyclotron with tools to reduce latency for applications where instructions have dependencies and cannot be eligible for execution based only on input data from the storage ring.

With the intention to create a system capable of scaling in the number of nodes and support workloads using huge data-sets, ownership request was introduced to have an homogeneous hot-set. Its integration contributes for a more precise definition of hot-set which is now refined to: a set composed of data chunks with the highest probability of utilization and optimal size for efficient network bandwidth usage, the highest throughput, and the lowest data access latency.

With this Chapter the Data Cyclotron foundation has been presented. The architecture (cf., Chapter 3) together with the hot-set management were conceptualized into a full functional system. The optimizations proposed in this Chapter and its integration with a DBMS are evaluated in the coming Chapter 5 using different hardware configurations and different workload scenarios.

Chapter 5

DBMS integration with the Data Cyclotron: DaCyDB

The Data Cyclotron is designed from the outset to extend the functionality of an application for intensive data analysis. The application accesses a Petabyte scale storage requesting data to be shared among a number of nodes to apply some filtering, aggregations, data-mining or execute machine learning algorithms. The the Data Cyclotron provides efficient access to the storage nodes by loading only the required data into the storage ring to feed starving computations.

The entire data set can be scanned for data processing, but the main advantage of using Data Cyclotron is the ability to specify one or more sub-sets of data from a single or multiple data sources and access them without a priory knowledge on how the data is distributed and its location at the computation nodes. There are several types of applications which can benefit from the Data Cyclotron data access model. For example:

- From a GPS database, a geographical zone is brought onto the storage ring to collect statistics, define new roads, or be correlated with other source of data, such as weather forecast, to search for patterns between weather state and traffic jams [102].

- Astronomers scanning a catalog from the space, such as SkyServer [137, 67, 82], in the search of new stars. The entire catalog is scanned but only one part of the space, at the time, is analyzed by specific algorithms.

- From a stream source, such as a stock market stream, different aggregations on previous windows are kept at the nodes and the new data is sent around to update the aggregation counters.
- Data mining on databases with a star schema. The fact tables are scanned once and distributed and cached among the nodes while the dimension data is sent around the ring to exploit data locality for some operators such as relational join [55, 54].

All examples have one thing in common, they benefit from the integration of a DBMS at each node. The application interacts with the Data Cyclotron and request data for local processing or simply express their computations through a frontend language, such as SQL, JDBC connector, or even directly through algebra operators. Those operators are then converted to optimized primitives part of a DBMS kernel.

In this Chapter we discuss the steps taken to integrate a column-store on a full functional Data Cyclotron, called *DaCyDB*. The evaluation of such an integration is done on two different computer clusters to study throughput and data access latency for different hardware configurations such as network bandwidth and main memory.

From the bottom to the top layer of the Data Cyclotron architecture, the Chapter identifies the challenges and issues to cope with different bandwidths, traffic jams, unbalanced data distribution, and a flexible query parallelism to exploit a continuous data stream. It explains the steps to be taken by the DBMS scheduler to have an effortless intra and inter query parallelism. Those steps are important for DaCyDB to scale up in the absence of knowledge about the number of nodes and data location. Such flexibility combined with iterative data loading and distributed consistency check is the key to reduce the cumulative cost to answer the first queries.

5.1 Outline

The Chapter is organized as follows. Section 5.2 presents the building blocks of MonetDB and its integration with the Data Cyclotron. Section 5.3 introduces the evaluation and the alignment of the implementation with observations done during simulation on Chapter 4. The evaluation is then split into two sections with each focusing on a different layer, network layer and application layer. Section 5.4 focus on how to have a symbiotic relationship between the network layer and the cluster for low data access latency and high throughput. Section 5.5, through a well known benchmark for data analysis (TPC-H), studies and explains the DBMS interaction with the bottom layers to explore efficient query parallelism. Finally, the Chapter is concluded with a summary in Section 5.6.

5.2 Towards a real system

A fully functional prototype is constructed to complement and exercise the protocols designed and analyzed through simulation. The DaCyDB prototype was realized by integration of the software components with MonetDB [1] since its inner workings are well known and its developers provided guidance on the internals of the system.

Its basic building blocks, its architecture, and its execution model are briefly reviewed using a SQL:2003 example [2]. The same example is then used to exemplify how MonetDB interacts with Data Cyclotron to fetch the data from the storage ring.

5.2.1 The MonetDB architecture

MonetDB is a modern fully functional column-store database system [105, 21, 22]. It stores data column-wise in binary structures, called Binary Association Tables, or BATs, which represent a mapping from an OID to a base type value. The storage structure is equivalent to large, memory-mapped, dense arrays. It is complemented with hash-structures for fast look up on OID and attribute values. Additional BAT properties are used to steer selection of more efficient algorithms, e.g., sorted columns lead to sort-merge join operations.

The software stack of MonetDB consists of three layers. The bottom layer, the kernel, is formed by a library that implements a binary-column storage engine. This engine is programmed using the MonetDB Assembly Language (MAL). The next layer, between the kernel and front-end, is formed by a series of targeted query optimizer. They perform plan transformations, i.e., take a MAL program and transform it into an improved one. The top layer consists of front-end compilers (SQL [3], XQuery [4], SciQL [91], Jason [23], etc.), that translate high-level queries into MAL plans [5].

The SQL front-end is used to exemplify how a MAL plan is created. An SQL query (cf., Figure 5.1) is translated into a parametrized representation, called a query template, by a factoring out its literal constants. This means that a query execution plan in MonetDB is not optimal in terms of a cost-model, because range selectivity do not have a strong influence on the plan structure. They do, however, exploit both well-known heuristic rewrite rules, e.g., selection push-down, and foreign-key properties, i.e., join indices. The query templates are kept in a query cache.

[1] For the integration, we added a new optimizer to the optimizers stack, called *opt_datacyclotron* available in MonetDB release Oct2010

[2] The system including our extensions can be downloaded from http://www.monetdb.org

[3] http://www.sql.org/

[4] http://www.xquery.com/

[5] http://www.monetdb.org/Documentation/Manuals/MonetDB/MALreference

```
SELECT c.t_id FROM t, c WHERE c.t_id = t.id;
```

Figure 5.1: SQL statement.

From top to bottom, the query plan in Figure 5.2a localizes the persistent BATs in the SQL catalog for ID and T_ID attributes using the bind operation. The major part is the binary relational algebra plan itself. It contains several instruction threads, starting at binding a persistent column (instruction 1) and 2) in Figure 5.2a), reducing it using a filter expression or joining it with another column [6] (instructions 3) to 6) in Figure 5.2a), until the results tuples are constructed. The last part constructs the query result table (instructions 7) to 9) in Figure 5.2a).

The query template is processed by an SQL-specific chain of optimizers before taking it into execution. The MAL program is interpreted in a dataflow driven fashion. The overhead of the interpreter is kept low, well below one μsec per instruction.

5.2.2 Plan generation

An SQL query is compiled into a MAL plan. This plan is then analyzed by the *opt_datacyclotron* optimizer, which injects three calls *request()*, *pin()* and *unpin()*. The *request()* identifies the required BATs. The *pin()* and the *unpin()* mark the time when a BAT is needed and subsequently released. They exchange resource management information between the application layer and the DaCy runtime.

The optimizer replaces each *BAT bind* call by a *request()* call and keeps a list of all outstanding BAT requests. For each relational operator argument, it checks if it comes from the Data Cyclotron layer. Its first utilization leads to injection of a *pin()* call into the plan. Likewise, the last reference of a variable is localized and an *unpin*() call is injected.

The code in Figure 5.2b is the MAL program from Figure 5.2a after being massaged by the *opt_datacyclotron* optimizer. It contains a *request()* call for the column *id* and another for the column *t_id*. Subsequently, the *pin()* call for the column *t_id* was inserted before the *bat.reverse()* operator, while for column *id*, it was before the first *join()* operator. Their *unpin()* calls are injected just after their last utilization. Note that such plan transformations are straightforward to integrate in a wide range of query optimizers and execution engines. Including the buffer manager of a traditional relational database engine.

[6]The reverse is an internal operator to invert the BAT for a join on the OIDs.

```
begin query();                    begin query();
1)  X1 := bind("t","id");           X2 := request("t","id");
2)  X6 := bind("c","t_id");         X3 := request("c","t_id");
                                    X6 := pin(X3);
3)  X9 := reverse(X6);              X9 := reverse(X6);
                                    unpin(X3);
                                    X1 := pin(X2);
4)  X10 := join(X1, X9);            X10 := join(X1, X9);
5)  X14 := reverse(X10);            X14 := reverse(X10);
6)  X15 := leftjoin(X14, X1);       X15 := leftjoin(X14, X1);
                                    unpin(X2);
7)  X16 := resultSet(X15);          X16 := resultSet(X15);
8)  X22 := io.stdout();             X22 := io.stdout();
9)  exportResult(X22,X16);          exportResult(X22,X16);
end query;                        end query;
```

(a) MAL structure (b) After the DaCy Optimizer

Figure 5.2: Selection over two tables.

5.2.3 Intra-query parallelism.

For large columns, and to exploit intra-query parallelism, MonetDB uses *partitioned parallelism* instead of *pipeline parallelism. Partitioned parallelism* is more suitable for the operator at the time paradigm used by MonetDB and to convert local execution plans to distribute execution plans. At the same time, relational pipelines are rarely very long. Furthermore, some relational operators do not emit their first output until they have consumed all their inputs such as sort and aggregate operators. Skewness is another issue since the execution of one operator is much greater than the others which limits the speedup of a pipeline [46].

Partitioned execution offers much better opportunities for speedup and scaleup. By taking the large relational operators and partitioning their inputs and outputs, it is possible to use divide-and-conquer to turn one big job into many independent little ones. This is an ideal situation for speedup and scaleup [46] and also to distribute query execution among a group of nodes.

With vertical partitioning imposed by the use of a column-store, horizontal partitioning is used to split large columns into **N** partitions [7]. The query plan is then

[7]These partitions are treated as normal BATs by the MonetDB kernel.

unrolled by repeating **N** times the code to process the original BAT. Such approach contrary to an iterative call within a loop allows MonetDB to explore cooperative work and the re-use of intermediates among queries. It helps the scheduler to be more flexible for any data arrival order. Furthermore, the repeated code can be wrapped into a function to then be called for each partition arrival. It gives the flexibility to have a mixture of push and pull operators within the same query plan. The approach has been explored with success by Pig [123] to define efficient data flow graphs for distribute query parallelism.

In the context of DaCyDB, depending on the DaCy buffer size, i.e., partitions size is at most equal to the DaCy buffer size, a column is partitioned into **N** partitions. Hence, instead of a bind be replaced by a single request, it is replaced by **N** requests. The injection of the calls is straight-forward. For example, Figure 5.3a depicts the plan of Figure 5.2b, but each column is now decomposed into three partitions.

The *opt_datacyclotron* optimizer injects **N** requests, **N** pins and **N** unpins into the query plan for each column. After the Nth *pin()* call the optimizer can insert a *pack()* call. This instruction does the union of all partitions rebuilding the original BAT. Alternatively, the pieces are kept independent to improve a parallel dataflow driven computation. Then a *pack()* instruction gathers the intermediates and glues them together to form the final result of an operator.

Figure 5.3a exhibits both cases: the instruction *reverse(X6)* from Figure 5.2b is parallelized while the instruction *join(X1,X9)* is not. This is possible due to a distinctive feature of MonetDB, intermediates materialization. It materializes the complete result of each operator.

Hence, parallel processing is improved by postponing *re-packing* as much as possible, i.e., the subsequent instructions are parallelized using the partitioned results. In Figure 5.3a, instruction **B** could also be parallelized by only doing the *pack* before instruction **C**. The unroll of the plan is taken care by the optimizer *opt_unroll* and its output is represented in Figure 5.3b.

For a blocking operation, such as *sort* and *resultSet* construction, the *pack* is mandatory for assembling the partial results again. For all other operators the plan can be unrolled as many times as possible to increase the degree of parallelism, however, we need to be careful to not have an exponential growth of the plan size. For example, the conversion of joins into Cartesian products or the unroll of multi-way joins. There is a plethora of alternative query plan optimizations, whose description and trade offs are discussed during the evaluation Section 5.5.

```
begin query();
1)  X2 := request("t","id",1);
2)  X3 := request("t","id",2);
3)  X4 := request("t","id",3);
4)  X5 := request("c","t_id",1);
5)  X11 := request("c","t_id",2);
6)  X12 := request("c","t_id",3);
7)  X17 := pin(X5);
8)  X18 := reverse(X17);
9)  unpin(X5);
10)  X19 := pin(X11);
11)  X20 := reverse(X19);
12)  unpin(X11);
13)  X21 := pin(X12);
14)  X22 := reverse(X21);
15)  unpin(X12);
16)  X9 := pack(X18,X20,X22);

17)  X24 := pin(X2);
18)  X25 := pin(X3);
19)  X26 := pin(X4);

20)  X1 := pack(X24,X25,X26);

21)  unpin(X2);
22)  unpin(X3);
23)  unpin(X4);

24)  X10 := join(X1, X9);          (A)
25)  X14 := reverse(X10);          (B)
26)  X15 := leftjoin(X14, X1);     (C)


27)  X16 := resultSet(1,1,X15);
28)  X23 := io.stdout();
29)  exportResult(X23,X16);
end query;
```

(a) After the opt_datacyclotron optimizer

```
begin query();
  X2 := request("t","id",1);
  X3 := request("t","id",2);
  X4 := request("t","id",3);
  X5 := request("c","t_id",1);
  X11 := request("c","t_id",2);
  X12 := request("c","t_id",3);
  X17 := pin(X5);
  X18 := reverse(X17);
  unpin(X5);
  X19 := pin(X11);
  X20 := reverse(X19);
  unpin(X11);
  X21 := pin(X12);
  X22 := reverse(X21);
  unpin(X12);
  X9 := pack(X18,X20,X22);
  X24 := pin(X2);
  X27 := join(X24, X9);
  unpin(X2);
  X30 := reverse(X27);
  X33 := leftjoin(X30, X24);
  X25 := pin(X3);
  X28 := join(X25, X9);
  unpin(X3);
  X31 := reverse(X28);
  X34 := leftjoin(X31, X25);
  X26 := pin(X4);
  X29 := join(X26, X9);
  unpin(X4);
  X32 := reverse(X29);
  X35 := leftjoin(X32, X26);
  X15 := pack(X33,X34,X35);
  X16 := resultSet(1,1,X15);
  X23 := io.stdout();
  exportResult(X23,X16);
end query;
```

(b) After the opt_unroll optimizer

Figure 5.3: MAL plan after opt_datacyclotron and opt_unroll.

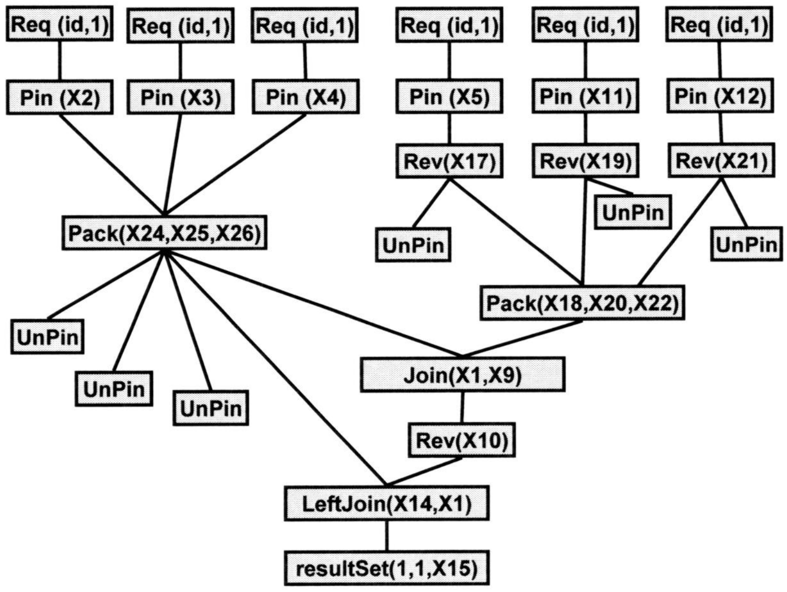


Figure 5.4: Execution graph.

5.2.4 Plan execution

The MAL plan is executed using concurrent interpreter threads, the *workers*, following dataflow dependencies. The dataflow dependencies are organized as an acyclic graph. Figure 5.4 contains the dataflow graph for the MAL plan in Figure 5.2b. At the top of the graph are the instructions which are only dependent on the load of data while at the bottom are the instructions dependent on the output of another instructions.

From the top left corner of the graph, all eligible instructions, this is, their input is available, are inserted into a queue **Q** as ready for execution. In *FIFO* order, a worker picks an instruction **I** for execution. Once **I** is executed the worker attempts to execute the next instruction which was waiting for **I**'s output to become eligible. If more than one, the others are added to **Q**.

The worker continues until it hits a blocking instruction, this is, an instruction that depends on more than one instruction output. This group of executed instructions is denominated as a *Dataflow* block. They can be executed independently from other instructions blocks. The first *Dataflow* blocks are defined around the binds, in DaCy context, around *request()* and *pin()* calls. To not block the worker thread, a *pin()* call is only set to eligible once the DaCy layer notifies the BAT arrival. This way a worker does not block while waiting for data (cf., Section 3.3.3).

" A request() *call is used to inform which data will be required. It is used by the DaCy layer to warm up local cache through a local or a remote data load. The* pin() *call is used to check the local cache for data availability. If it is not available, the application call blocks. ... The application call remains blocked until the data chunk is made available at the DaCy buffers. Once available, the application call resumes and uses the data chunk for processing "*

Based on the BATs arrival, a worker selects a *Dataflow* block for which a BAT is the input. It attempts to execute the entire block. A worker picks an instruction from the queue for which the output of the executed instruction is the input. For each instruction it does an *admission* check to see if there is enough resources to execute the instruction, this is, enough memory to store the result, and apart from *pin()* calls, have the input in memory. In case admission check fails for an instruction (**A**), the worker attempts the execution of instructions from other *Dataflow* blocks. In case its search does not succeed, it executes **A** with the risk of triggering another instruction results, or inputs, of being swapped out. The *admission* check intends to reduce virtual memory utilization and have balanced resource utilization, this is, a fair competition for them.

Combined with the inter-query parallelism, the scheduler uses the arrival of a single BAT to resume execution of a group of *Dataflow* blocks. Furthermore, there is not an explicit order to process the column partitions within each individual plan. Instead, the execution method is driven by partition availability, much like the approach proposed for *cooperative scans* [155], where a query can join the column scan at any partition.

With plans extended for partitioned parallelism we maximized the opportunities to exploit cooperative actions on the shared data chunks as they become available. Hence, the ratio between the buffer space and the data size for the workload is less than one, i.e., there is often more buffer space available to start newly arrived queries. The direct data access to the RDMA buffers, and the internal query parallelism with cooperative access to the partitions, increases the throughput and keeps the query response time low.

Such cooperative work is only possible due to the strategy taken by MonetDB to unroll the loops through code repetition (cf., Section 5.2.3) which combined with code re-utilization, such as recycling [84, 83], turns the overhead of defining and executing long plans negligible.

5.2.5 Query distribution

A balanced system, i.e., system without hot spots, contains an equal load over all nodes. Therefore, on query assignment, the node with ample resources or the smallest query-queue can be chosen to execute the query. However, for decentralized architectures determination of the right node is complex because the load of each node is not globally known.

In most architectures the query assignment is based on a global cost model. The cost model assigns the queries based on the data location. The success of load balancing is thus, dependent on the efficiency and precision of the cost model. However, a shift on the data key space used by the workload requires re-allocation of the data and a re-organization of the distributed infrastructure. The cost model precision is affected by those changes, or in worst case, it becomes invalid. Its adaptation is complex and temporary which makes distributed systems inflexible to fluctuations on the workload interest.

Exploiting the fact that each individual query can be processed at any node and inspired by an earlier attempt for a decentralized query assignment which achieves load balancing, Mariposa [134], the DaCyDB leaves the decision to execute a query to the individual nodes. There is no global cost model for query assignment which makes the system flexible to scale and to exploit all available resources.

In *Mariposa* [134] an economic model was proposed for query assignment. In this model, the decision to execute the query is left to the individual nodes. In [134] a node advertises its services, buys objects from another sites and bids on queries. Each query contains an offer for different execution times. A node, based on the local resources, *bids* on queries with the goal to maximize its profit.

The DaCyDB, a decentralized distributed system, embarks on the same spirit, i.e., a query it is not implicit assigned to a node. Each node collects queries from a common queue modeled as the storage ring. Unlike *Mariposa*, there is no bidding process, the nodes simply select the queries with the highest priority, in a *first-come-first-served* approach, *FIFO*.

The priority of a query is increased with each transverse of the ring and not be picked for execution. A priority based on the number of cycles works as a timeout for their stay in the ring and for their execution in the nodes. Each node selects the queries with priority within a certain range, i.e., the *hottest* queries in the ring. In case of lack of resources, the node postpones the selection until further release of resources.

The queries are flowing in the ring counter clockwise. Having the queries flowing around the ring, a distributed FIFO queue is defined. It assures an equal distribution of queries among the nodes, this is, all nodes are used to process the queries even if the user has selected a single node to submit the queries.

5.3 Evaluation

The primary goal of this evaluation is to identify advantageous scenarios and possible bottlenecks of the integration of MonetDB with the Data Cyclotron. The experiments are not intended to benchmark our novel architecture against contemporary approaches of single/parallel solutions. The engineering optimization space for the Data Cyclotron is largely unexplored and experience gained to date with the architecture would not do justice to such a comparison.

The experiments are intended to align the full-functional system with the one used for simulation. They are intended to study each layer in detail by disclosing issues not identified during simulation and for such issues propose solutions.

Starting with the bottom layer, the network layer, micro-benchmark experiments are used to show the influence of the hardware on the data access latency such as network and memory bus type. We study how the ring structural organization and number of queries per node affects latency and throughput. Furthermore, the cache policies induce the hot-set size reduction by avoiding the circulation of *unpopular* data and data re-loads.

Consequently, we focus on the application layer to study query parallelism. The experiments intend to understand why the actual query execution model used by MonetDB is not yet optimal to exploit the advantages of the Data Cyclotron. The operator selection and intermediate results management are the points stressed. The experiments resume to an analysis of the system throughput using a full-scale TPC-H implementation with workloads of thousands of queries and different amount of resources.

5.3.1 Clusters

The evaluation uses two computer clusters. They represent the Present and Future for data centers. Their difference helps us to test the Data Cyclotron robustness and efficiency for current data centers, and the feasibility of the architecture for future state-of-the-art data centers.

Cluster 1, from our national super-computer center [8], represents a common HPC cluster used for eScience. It is equipped with 256 machines, all of them interconnected using a Qlogic 4x DDR InfiniBand card, i.e., the bandwidth is 1600 MB/sec and the latency below 6 μsec. Each machine is equipped with 2 Intel quad-core Xeon L5520 (2.26GHz clock), 24 GB main memory, 8 MB cache size, 65 GB scratch space (200 MB/sec read access), PCI bus 2.0, and 5.86 GT/s (GigaTransfers per second)

[8]http://www.sara.nl/

quick path interconnect. Each rack has two switches where each of them has 16 nodes directly interconnected. Virtual memory is defined as a partition in the scratch space.

Cluster 2 is a dedicated 300 database machine cluster from the Scilens [128] project. It is a state-of-the-art cluster exploiting new trends in networking, 40 Gb InfiniBand at the edges and 100 Gb InfiniBand between switches. The cluster is composed of two tiers of machines. The two tiers organization approximates the cluster to the organizational structure of major data centers such as Google, Microsoft Azure, Yahoo, etc., this is, storage nodes separated from the computational nodes, (cf., Section 2.4).

" A server is not anymore a single box with all compute resources in it. They have been placed in pools together with storage and memory processors. All components have been spread around the data center and they are all interconnected through a network. ... Cloud computing platforms such as Windows Azure, Yahoo storage, or Google storage, have their storage services separated from compute nodes "

The bottom layer of machines, *the pebbles* (aka storage nodes), consists of 144 shoeboxes with an AMD Bobcat, 8 GB RAM, 10 Gb Ethernet, and 5 2 TB HDD configured as 4 in RAID0 and one system disk. The next layer, *the rocks* (aka computational nodes), consist of 144 shuttle boxes with 2 Intel quad-core K2600 (3.40GHz clock), 8 MB cache size, 16GB main memory, PCI bus 2.0, 40 Gb InfiniBand, and a 2 TB HDD (145 MB/sec read access). Each rack has two switches where each of them has 18 nodes directly interconnected. Like *Cluster 1*, virtual memory is defined as a partition in the scratch space.

5.4 Network layer

The base topology used in this experiment is a ring composed of thirty-two nodes from *Cluster 1*. The nodes are spread among three switches installed into two racks, rack $\mathbf{A}$ and rack $\mathbf{B}$. The nodes are interconnected in the same order as the one represented in Figure 5.5.

All nodes dedicate one third of their memory for RDMA buffers, 8 GB for *DaCy storage*, leaving enough memory space for intermediate query results, this is, 16 GB. Hence, the query performance is not affected by memory BUS contention, nor being bounded by the local disk bandwidth in case of swapping.

The micro-benchmark data-set consists of three 64-column wide tables, TA, TB, and TC. They differ in the number of partitions (data chunks) per column: TA uses 2 partitions, TB 3 partitions, and TC 6 partitions. Each partition has 128 MB size to optimize network transport, leading to a total database size of around 270 GB.

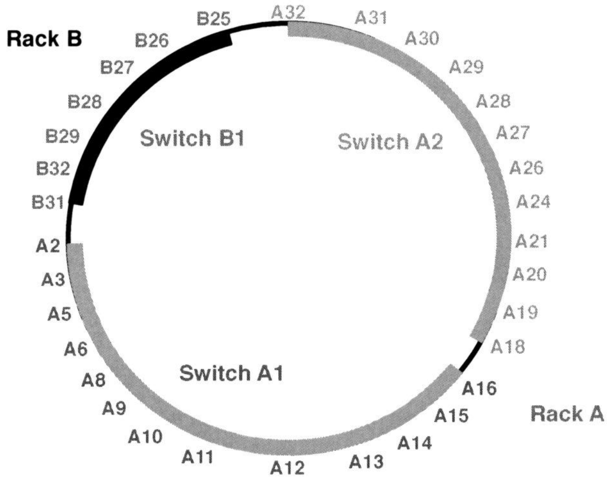


Figure 5.5: Ring structure.

The micro benchmark workload is divided into 6 scenarios, $S1, .., S6$. Each contains 1024 simple select queries over 11 randomly chosen columns, 4 joins and a projection on 3 columns all from the same table. The first two scenarios, $S1$ and $S2$ use TA, TB is used by $S3$ and $S4$, and TC is used by $S5$ and $S6$. To have different hot-set sizes, $S1$, $S3$, and $S5$ only touch on half of the columns, while the remaining scenarios, $S2$, $S4$, and $S6$, touch on all columns.

Furthermore, to stress the Data Cyclotron buffer pool for data access and forwarding, each query keeps 70% to 90% of the requested data in the buffers, i.e., DBMS space load, and the $LOIT$ is kept as low as possible to maximize the amount of hot-set data in the ring, i.e., the DaCy space load. Hence, from scenario 1 up to 6, the DBMS space is increased shrinking the DaCy space, i.e., the available space to store the hot-set. All workloads are executed with a low $LOIT$ with the exception of $S6$ where $LOIT$ reached the top levels due to the lack of space in the storage ring to accommodate the hot-set, which aligns with the observations in Section 4.3.1.

The data is distributed among the 16 nodes, 7 from rack B (within the same switch) and the remaining ones from rack A (within the same switch), see Figure 5.5. The un-balanced distribution is used to study tension on the node's buffers for data loading, forwarding, and processing. On the other hand, the queries are flowing counter clockwise in the ring and nodes automatically pick and execute one query at a time, this is, a balanced distribution of their execution among the nodes.

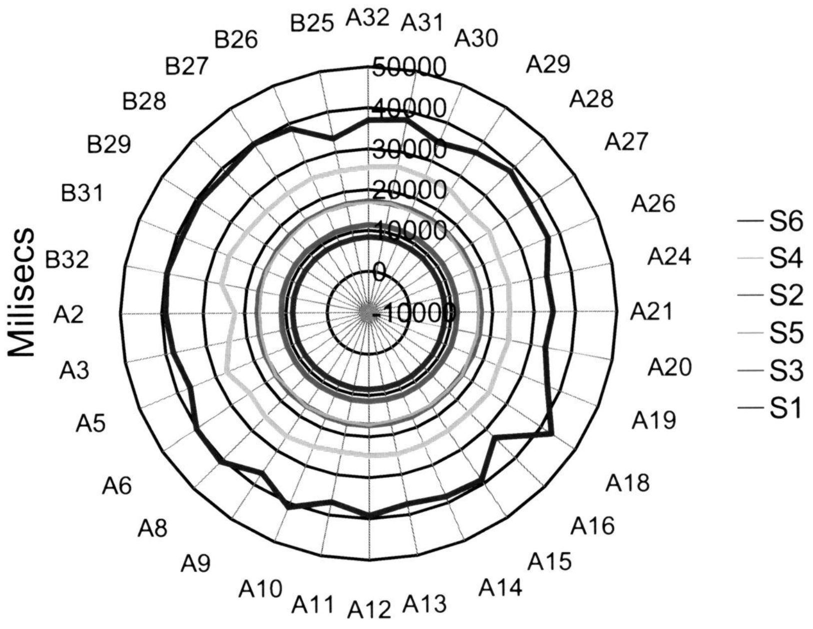


Figure 5.6: Query average time per node.

5.4.1 Bandwidth

The first experiment aims to study the relation between the hot-set size, bandwidth, and query average response time. For the six scenarios the query execution time average is pictured in Figure 5.6. For different hot-set sizes, the Data Cyclotron was able to provide the required data for the queries at high speed. The average time increase is close to linear for the ones with the low $LOIT$, i.e., the complete hot-set is kept in the ring.

Despite buffer contention for data access and forwarding, the data stream achieved 1.3 GB/sec in the best cases and 0.9 GB/sec in the worst cases. In the same order of magnitude the nodes were receiving data, this is, bi-directional traffic was 2.6 GB/sec for best cases and 2.1 GB/sec for worst cases [9]. The measurements are the average time

[9]The 1.8 GB/sec would be worst case if a node, at same time, sends and receives data from a different rack which is not the case.

Protocol	bw_mode	min_bw	max_bw
tcp	uni-directional	0.50 GB/sec	0.90 GB/sec
rc	uni-directional	1.40 GB/sec	1.60 GB/sec
rc	bi-directional	2.98 GB/sec	3.15 GB/sec

Table 5.1: Cluster 1 - Bandwidth measurements.

to receive and forward 100 partitions of size 128 MB.

Compared to isolated measurements done through a benchmark to measure RDMA and IP performance, qperf [61], the achieved bandwidth is not far from optimal. On Table 5.1 [10], are the measurements using *qperf* for hundred messages (-n 100), of size 128 MB (-m 128M), using a *MTU* of size 2k (-mt 2048) and with polling mode [11] off (-cp 0).

The benchmark results are near to the theoretical optimal of the network hardware confirming an excellent network configuration. However, for a non-isolated scenario like the Data Cyclotron, i.e., the memory bus and the CPU are also busy with data processing, achieving the optimal bandwidth is not always possible. Furthermore, without a monitoring tool for the asynchronous communication of RDMA at *Cluster 1*, the reported times are not isolated measurements. They contain some noise such as the latency of the signals to acknowledge the received data and selection of the next buffer.

The results demonstrate how well the buffer space is managed between the Data Cyclotron and the DBMS. Unfortunately, the 1.3 GB/s bandwidth is only achieved within the same hardware rack and network switch. The ratio between the hot-set size and the query execution time average shows that for each extra GB of hot-set data, the execution time of a query increases between 750 milliseconds and 950 milliseconds, i.e., it is bounded by the network bandwidth.

The inherent data flow, i.e., the intra-query parallelism, makes possible to access all data needed within a single DaCy-cycle. Hence, for I/O bounded workloads the query execution is bounded by the hot-set size, network bandwidth and buffer space. Hence, the data access latency is the major cost in the response time. The Data Cyclotron is, however, intended for heavy and complex analytic queries composed of instructions with execution time reaching a few hundreds of seconds. This creates an interval where data access overlaps with data processing and, due to RDMA, without taking CPU cycles.

[10]TCP is the acronym for TCP socket while RC for Reliable Connection protocol for RDMA.

[11]Relevant to the RDMA tests and determines whether they poll or wait on the completion queues. If *OnOff* is 0, they wait; otherwise they poll.

5.4.2 Traffic jams

The experiment in Section 5.4.1 attempts to study the consequences of traffic jams in the query average time. Traffic jams, also known as traffic congestion, is a condition on networks that occurs as use increases or at some point the link capacity shrinks. It is characterized by slower speeds and longer queues. With a ring configured as in Figured 5.5 and bandwidth variation when crossing a switch, or a rack, traffic jams will certainly happen.

Traffic jams are only visible for a large ring. For large rings only a sub-set of nodes has the data forwarding slowed down while on others it flows at normal speed without any congestion. In small rings, after a certain period of time, due to the traffic jam backwards propagation, all nodes become overloaded, therefore, they all forward data at the same speed. The traffic jam propagation depends on congestion at the nodes buffers. Higher the number of nodes with congestion far away goes the traffic jam propagation. As a result, the data access latency increases.

In Figure 5.6 for scenario $S6$, a traffic jam propagation s visible among the sub-set of nodes located between $\mathbf{A12}$ and $\mathbf{B25}$. With the nodes located in two different racks, $\mathbf{A}$ and $\mathbf{B}$, and all of them loading data, the traffic jam from the *edge* nodes [12] is spread among their neighbors. The traffic jam propagation is due to the tension on the their buffers. It quickly synchronizes the data forwarding speed of the entire sub-set of nodes.

The nodes on the other half of the ring do not have traffic jam propagation. Traffic only slows down at the edge nodes. A clear example is $\mathbf{A18}$. It has higher query average time compared to its neighbors.

Nodes receiving data from edge nodes are the ones with highest number of empty buffers. They send data faster than they receive. Hence, no cache clean is triggered and data chunks have always space to be kept for execution. For this reason, the query response time at these nodes tends to be lower than the other nodes. A good example is node $\mathbf{A16}$.

On all other nodes, the synchronization of data forwarding imposes a similar query average time among the nodes, with few exceptions. For node $\mathbf{B25}$, it was expected an increase of the response time due to the amount of data loaded by the preceding nodes and for being an edge node. However, Figure 5.6 shows the opposite. The reason for this exception comes from the average time to fulfill requests. $\mathbf{B25}$ has the best average time among all nodes.

For workloads like $S6$, and a small storage ring, not all data chunks can co-exist at the same time in hot-set. Hence, their re-load is constantly requested. With requests

[12]A edge node is a node which communicates with a node from another switch.

Protocol	bw_mode	min_bw	max_bw
tcp	uni-directional	0.90 GB/sec	1.20 GB/sec
rc	uni-directional	3.00 GB/sec	3.10 GB/sec
rc	bi-directional	4.00 GB/sec	4.03 GB/sec

Table 5.2: Cluster 2 - Bandwidth measurements.

flowing counter clockwise, data clockwise, and data loaded in half of the ring preceding **B25**, **B25** has the best location for the lowest average time for data access latency. On the other hand, **A12** has the worst location.

Does a wider link reduce the buffers tension and eliminate traffic jams?

Cluster 2 is equipped with *QDR* InfiniBand (40 Gb/sec signal rate) instead of *DDR* InfiniBand (20 Gb/sec signal rate). A wider network link since it has 4 GB/sec data rate in theory. However, in practice the real date rate speed is always below the theoretical value. To determine the real uni- and bi- directional bandwidth, we have used once again *qperf* benchmark to determine the real link bandwidth. The results are summarized in Table 5.2 [13].

Contrary to *Cluster 1*, the bi-directional bandwidth is not twice the uni-directional bandwidth. After some investigation and isolated benchmarking, the memory bus slots and its version, i.e., PCI bus 2.0, were identified as the reason for such low bi-directional bandwidth. The nodes composing *Cluster 2* are commodity desktops which have PCI bus 16x slots to plug graphic cards. Such slots have asymmetric bandwidth contrary to the 8x slots used for servers. The same benchmark on two independent servers showed a bi-directional bandwidth of 6.52 GB/sec while the uni-directional bandwidth was 3.39 GB/sec.

The bi- and uni- directional bandwidth for the two independent servers is an indication that, even with proper slots, the PCI bus 2.0 is the bottleneck for remote data access instead of the QDR network link. The expected results are 3.8 GB/sec uni-directional bandwidth and 7.6 GB/sec bi-directional bandwidth. Such values are only possible with PCI bus 3.0.

The authors in [98], for a smaller scale problem and with nodes wired back to back, this is, without a switch, have studied the bandwidth differences between the two PCI bus versions. The work shows why the CPU, chipset and the memory bus are three important factors to achieve the optimal bi-directional bandwidth. Their evaluation

[13]TCP is the acronym for TCP socket while RC for Reliable Connection protocol for RDMA.

Scenario	Time (sec)
S2	14.40
S4	18.73
S6	28.49

Table 5.3: Cluster 2 - Average response time per query.

suggested that PCI bus 3.0 is the one to be used to achieve the maximum bi-directional bandwidth. Not only for its wider links, but also for the encoding used. The encoding used by PCI 2.0 is the 8B/10B (every 10 bits sent carry 8bits of data) instead of 64B/66B (every 66 bits sent carry 64 bits of data) used by QDR. For the later the overhead is 3.125% while for 8B/10B encoding has 25% overhead.

With 2.0 GB/sec as the maximum bandwidth achieved in each direction, *Cluster 2* only offers 1 GB/sec extra bi-directional bandwidth compared to *Cluster 1*. To determine how far is the bandwidth achieved by Data Cyclotron from this isolated measurements, the micro-benchmark was re-run for 16 nodes within the same switch and 32 nodes spread among two switches.

The bi-directional bandwidth achieved in both rings was near to 4.0 GB/sec among all nodes and no traffic jams were detected. With PCI bus 2.0 and asymmetric bandwidth, the links are never saturated, therefore, there is not traffic jams at the edge nodes. Without traffic jams and higher bandwidth, the tension on the nodes buffers is also reduced contributing for lower query response time. The result in Table 5.3 confirms that. Using only the three scenarios of 64 columns, this is, $S2$, $S4$, $S6$, we see an improvement of factor 1.3, 1.5, and 1.3 [14].

5.4.3 Dynamic readjustments

To reduce tension on nodes buffers and data access latency, ownership delegation was introduced for clusters like *Cluster 1*. With ownership delegation, a node is now allowed to delegate some data chunks ownership when an high $LOIT$ does not reduce the tension on its buffers.

In the context of the scenario from previous Section 5.4.1, the *groovy* nodes [15] would tag a sub-set of the most frequently loaded chunks to be unloaded by *non-groovy* nodes. Hence, nodes on the right side of the ring, Figure 5.5, would load data chunks

[14] Factor 1.2 if compared with the right side response times and factor 1.4 if compared with left side response times.

[15] Sub-set of nodes from which most of the data is requested.

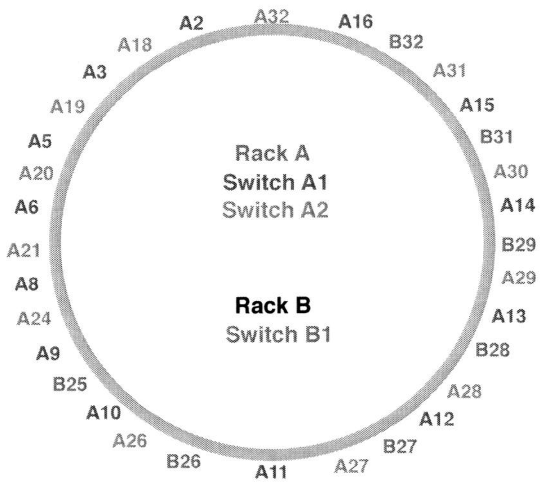


Figure 5.7: New ring structure.

with the tag $Delegated$, so nodes on the left side of the ring would take over their ownership. The ownership delegation is used to reduce traffic jams created by congestion on DaCy storage for data loading and forwarding. The readjustment is independent of the application layer and it is all conducted by the DaCy layer.

Ownership delegation is, however, not enough to avoid traffic jams at the *edge* nodes. For this situation, the bandwidth in all nodes needs to be re-adjusted to the lowest level [16]. Hence, in *Cluster 1* the inner switch forwarding speed should be equal to the forwarding speed at the *edge* nodes, i.e., homogeneous bandwidth among all nodes.

The adjustment can be done at hardware level or at the software level. At hardware level, nodes from the left side of the ring, from rack **B**, would be mixed with the nodes from rack **A**. Such re-organization would impose to the data flow a lower data forwarding speed due to the frequent hops between both racks, see Figure 5.7.

At software level, the adjustment would be based on variations of the DaCy-cycle duration and collection of statistics. However, the precision of decisions based only on these two factors would be lower than the ring re-organization. Another reason for a physical re-organization of the ring is to have the same average of hops to request and access a data chunk on all nodes. In the coming Chapter 6 the dynamic adjustment

[16]The general concept is highly used on traffic control on high-ways, this is, it is better to move at a lower speed than be blocked in a traffic jam.

of resources is introduced to envision a more flexible architecture to cope with load balancing challenges.

5.4.4 Adaptive nodes

A the Data Cyclotron ring is composed of nodes with autonomous and dynamic behavior towards the workload on the system. Each node tries to accommodate as many queries as possible exploiting its resources and thus improve throughput. Hence, each node dynamically adjusts the number of internal clients to pick queries for execution, i.e., *pickers*, and the number of DaCy buffers to store application data.

A picker is a thread that picks a query, sends it for execution and returns the result. The query in compiled into a set of *Dataflow* blocks, i.e., group of instructions, which compose the query execution plan. The instructions composing these blocks are inserted into a queue $\mathbf{Q}$ for execution.

On the other side of the queue are the workers, threads that execute *Dataflow* blocks. Based on the arrival of a BAT, a worker selects a block for which the BAT is the input data. It attempts to execute the full block and once done, it moves to the next block which has input data available (cf., Section 5.2.4).

The initial number of *pickers* is one while the number of *workers* is determined by the number of cores of a node. A node with $\mathbf{N}$ cores allocates one core for DaCy routines and the remain ones, $\mathbf{N}-\mathbf{1}$, for workers [17].

The number of workers is constant, however, the number of pickers increases or decreases over the time based on the workload characteristics. For I/O bounded workloads, the number of pickers is increased to explore cooperative work between them, i.e., different blocks of instructions can use the same input data. On the other hand, for CPU bounded workloads the number of pickers is kept as low as possible to have less instructions competing for the available CPU cycles.

For the micro benchmark scenarios, the workloads are I/O bounded. Using *Cluster 2* and scenarios $S2$, $S4$, and $S6$, the number of pickers is increased step wise by one picker until the queries get blocked. Queries get blocked when the application space together with the neutral zone is not enough to store all required data chunks to proceed with the execution of *Dataflow* blocks (cf., Section 3.5.1).

" The DaCy storage is divided into the DaCy space *and* application space. *A low number of free buffers for* transit data *slows down the data propagation, therefore, it decreases the global throughput. On the other hand, few buffers to store data for the application degrades the performance of local computations. ... The DaCy sets a min-*

[17]The thread assignment to the cores is left to the Operating System.

imal number of buffer space for each type of data. The remaining space, neutral zone, *is used for both data types depending on the workload requirements "*

On the first attempt, for $S2$ and $S4$ the queries got blocked once the number of pickers was increased to three. For $S6$, queries got blocked for two pickers. With all *requests* getting eligible at query initialization time, the *pin()* calls from all queries had to compete against the scarce number of buffers available. For example, each $S6$'s column has six partitions, with queries with low number of dependencies, almost all *pins* become eligible. Hence, several *Dataflow* blocks are initialized. Each of them has input data which is only released once its output is used by the preceding operator, often a blocking operator. During its processing, views are used for efficient utilization of memory for intermediates. Therefore, if the blocking operator is waiting for the output of another *Dataflow* block which does not have free DaCy buffers to pin its input data, the query execution gets blocked.

To solve the issue, the *opt_datacyclotron* optimizer was improved to reduce the number of requests eligible for execution. Instead of making all requests eligible, the optimizer makes **X** requests eligible while the remain ones become eligible as pin calls are executed. With the new optimizer version, the MAL plan from Figure 5.3b is now replaced by the MAL plan from Figure 5.8.

To control the eligibility of a *request()* a new input argument was introduced. For the first requests the input value comes from the instruction *init()* [18]. It is the first one to be executed and sets the first **X** requests eligible for execution. In this example, **X** has value two. The remain *requests* use the output of remaining *pin()* calls to become eligible. For example, the request represented by instruction $B)$ is only eligible for execution once the *pin* represented by instruction $A)$ is executed.

Using the new version of *opt_datacyclotron*, we re-run the queries using different number of pickers. Different values of **X** were used to see how this configuration parameter affects query average response time. The values of **X** were chosen based on the maximum number of data chunks requested in each scenario. For $S6$ each columns has six partitions which means each query requests 66 different data chunks. For $S2$ a column has two partitions, therefore, only 22 data chunks are requested per query. The most relevant results are picture in Table 5.4.

Reducing the number of eligible requests, the query response time increases. The more partitions a column has, the higher is the impact in the query response time. Nevertheless, the reduction of eligible requests reduced the number of buffers in the application space. To exploit this extra space to store input data we increased the number pickers step wise from one to four pickers for each scenario and for each value

[18]The BAT is used as input only for eligibility control and not for processing.

```
begin query();
  X1 := init();
  X5 := request(X1,"c","t_id",1);
  X11 := request(X1,"c","t_id",2);
  X17 := pin(X5);                              (A)
  X12 := request(X17,"c","t_id",3);            (B)
  X18 := reverse(X17);
  unpin(X5);
  X19 := pin(X11);
  X2 := request(X19,"t","id",1);
  X20 := reverse(X19);
  unpin(X11);
  X21 := pin(X12);
  X3 := request(X21,"t","id",2);
  X22 := reverse(X21);
  unpin(X12);
  X9 := pack(X18,X20,X22);
  X24 := pin(X2);
  X4 := request(X24,"t","id",3);
  X27 := join(X24, X9);
  unpin(X2);
  X30 := reverse(X27);
  X33 := leftjoin(X30, X24);
  X25 := pin(X3);
  X28 := join(X25, X9);
  unpin(X3);
  X31 := reverse(X28);
  X34 := leftjoin(X31, X25);
  X26 := pin(X4);
  X29 := join(X26, X9);
  unpin(X4);
  X32 := reverse(X29);
  X35 := leftjoin(X32, X26);
  X15 := pack(X33,X34,X35);
  X16 := resultSet(1,1,X15);
  X23 := io.stdout();
  exportResult(X23,X16);
end query;
```

Figure 5.8: MAL plan after being unfolded by new *opt_datacyclotron*.

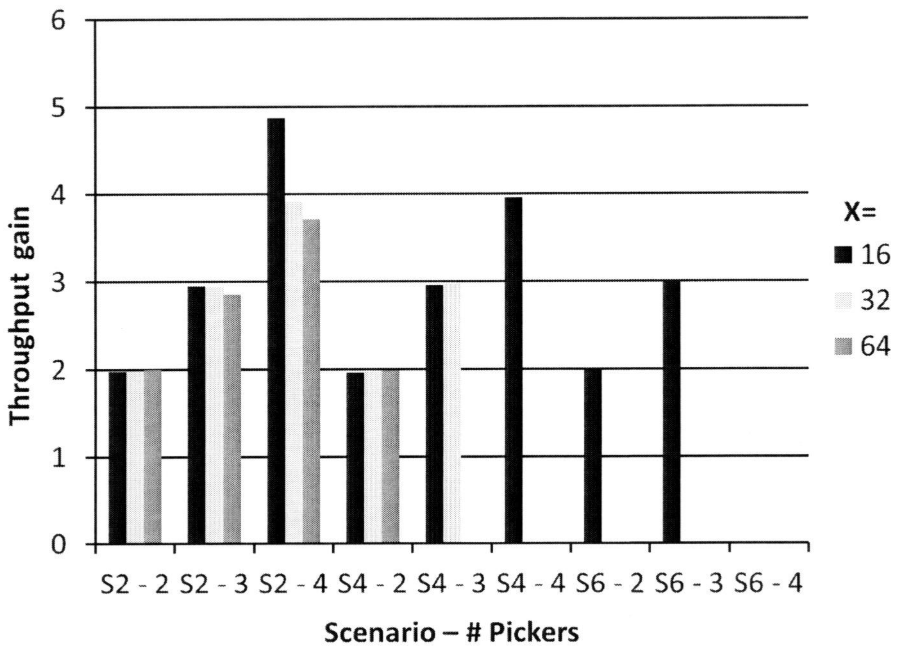


Figure 5.9: Throughput gain compared to one picker.

of **X**.

For scenarios with high number of partitions, such as $S6$, the reduction of eligible requests has to be aggressive if we want to have more than one picker, and thus increase throughput. In Figure 5.9 is the relative gain in throughput per node compared to the one picker results from Table 5.4.

X	S2	S4	S6
64	14.6 (sec)	18.7 (sec)	28.5 (sec)
32	14.7 (sec)	32.2 (sec)	50.2 (sec)
20	19.5 (sec)	35.3 (sec)	63.2 (sec)
16	25.6 (sec)	41.5 (sec)	72.1 (sec)

Table 5.4: Average response time per query for controlled eligibility.

The increase of throughput is evident. The main reason for its increase is the direct access to data from the DaCy buffers. Those data chunks are in-use, or were just used, by the other local queries. The higher the number of pickers, the higher is the probability to have the requested data in local DaCy buffers.

The difference for the throughput gain between each scenario comes from the increase of competition for the empty buffers to store input data. The competition is directly proportional to the number of data chunks per query. Such competition is identified by the query response time standard deviation. For $\mathbf{X}$ equal to 32, $S4$ had some query execution times above 80 seconds for three pickers and queries getting blocked four pickers. In scenario $S6$, few queries crossed the line of the 100 seconds for two pickers and get blocked after three or more pickers. Despite the increase of query response time, the throughput increased even more. The highest gain goes for $S2$, a factor five improvement while query response time decreased to 1.8 of optimal. For optimal results, the scheduler should give priority to the requests which are requesting data chunks already in the hot-set or in cache.

This experiment shows how local resources can be exploited to improve throughput. Through a configuration variable the user can now trade query response time for higher throughput and vice-versa. The number of pickers is then dynamically adjusted to avoid data starvation. For complex queries, the variable is used to control the virtual memory usage. The study of its benefits, in the context of complex queries, is postponed to Section 5.4.4.

5.4.5 Ring extension

Any system configuration faces the point that it lacks resources to accommodate a timely execution of all queries. In the Data Cyclotron this can be partly alleviated using a ring extension. With the same setup, the ring on *Cluster 1* is extended twice through insertion of 16 nodes: $R1$, $R2$, and $R3$.

Global throughput.

The results of this experiment are shown in Figure 5.10. For each scenario the average time for each query remains almost the same as for the first extension. In the second extension the average time over all scenarios increased between 10 and 16%. With up to 48 nodes, the throughput per node increased slightly. However, for rings beyond 64 nodes we reached a saturation point and the throughput per node decreased by 13%. The length of the ring lead to an higher data latency, but since the hot-set size did not change, and with an high bandwidth being available, the system throughput increased by 70% when 16 nodes were added. Addition of another 16 nodes only

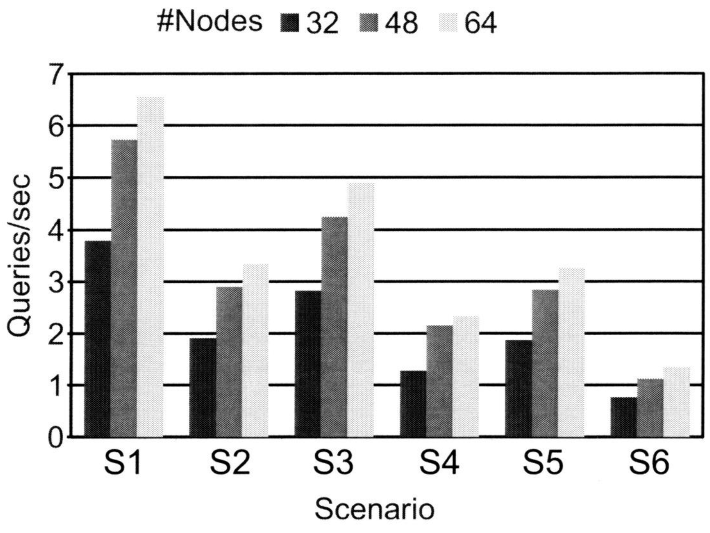


Figure 5.10: System throughput.

brought 14% improvement (except for $S5$). It indicates that the *saturation point* is reached by another extension of 16 nodes.

The same experiment on *Cluster 2* showed the saturation point is only achieved with a slightly larger ring. With a faster network link, the traffic jams due to tension on the node's buffers became almost un-noticed for large rings. It promotes the idea that the Data Cyclotron becomes more efficient for data centers with higher bandwidths than the ones used for these experiments. Hence, with the road-map for network hardware the Data Cyclotron can use rings over multiple racks with low deterioration of the global throughput.

Throughput per node.

The number of data chunks per scenario is summarized on Table 5.5. Knowing that queries are network I/O bound, the throughput per node is inversely related to the hot-set size, Figure 5.11. For example, $S2$'s hot-set size is two times bigger than $S1$'s hot-set, the throughput per node for $S2$ is two times smaller than $S1$. The same correlation between the hot-set size difference and the throughput per node, with some

Scenario	#columns	#partitions	#data chunks
S1	32	2	64
S2	64	2	128
S3	32	3	96
S4	64	3	192
S5	32	6	192
S6	64	6	384

Table 5.5: Number of data chunks for each scenario.

noise, holds for $S3$, $S6$, and $S4$. The noise is related to the number of partitions per column. The more partitions a column has the easier it is for MonetDB to exploit intra-query parallelism.

The different number of partitions raises also another interesting observation. Comparing $S5$ and $S4$, $S5$ hot-set has the same number of data chunks as $S4$, however, $S5$'s throughput per node is similar to $S2$ instead of $S4$. Queries in $S5$ despite requesting

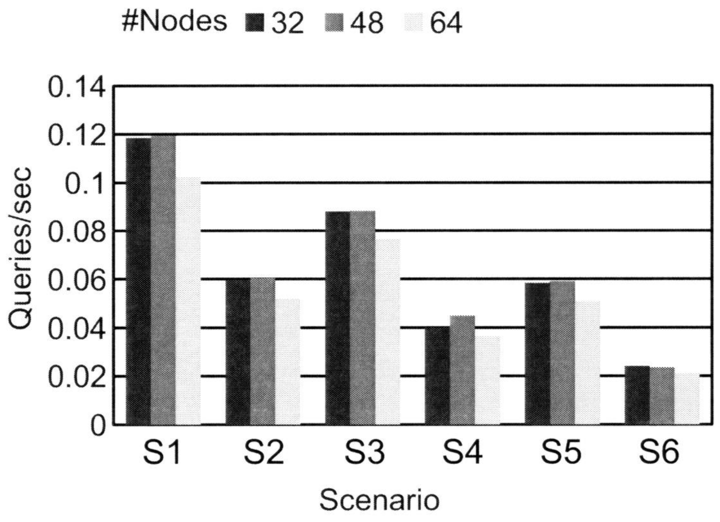


Figure 5.11: Throughput per node.

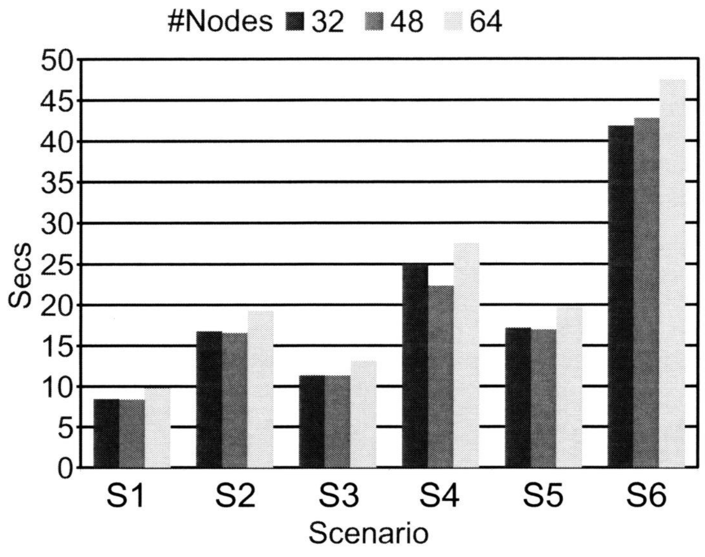


Figure 5.12: Query execution average per ring.

more data than $S4$, they request the same number of columns as $S4$. Since partitions for a column tend to be loaded within the same DaCy-cycle and travel together, or at least, are seen within the same DaCy-cycle, $S5$ queries have higher probability to fulfill a sequence of requests within a DaCy-cycle.

The difference is emphasized in this context due to the query simplicity which allows perfect intra-query parallelism without access to the virtual memory to store intermediates. For more complex queries, the dependencies between instructions might reduce this gain in throughput. The same might happen on nodes with lack of resources to store intermediate results. Those scenarios and issue are studied in detail in Section 5.5 using a well known benchmark for data analysis, TPC-H, and nodes with different memory size.

Query response time.

For all scenarios, with the exception of $S6$, the query execution average had the lowest value for the 48 nodes ring, Figure 5.12. The storage ring provided by 48 nodes is capable to keep most of the hot-set in rotation. Hence, the latency added by 16 extra nodes is fully absorbed by the average data access time. With the required data chunks flowing around, smaller is the average time to request them.

For $S6$, the same effect is only visible with an extension lower than 16 nodes. The tension at the *groovy* nodes buffers, keeps the average time data access still high which avoids the extra latency imposed by the new nodes to be absorbed.

For all scenarios the optimal number of nodes, for the highest throughput while keeping the same query average time is thus, only identifiable with a step wise ring extension. The nodes should be added one at the time, or in small sets. The new nodes should enter the ring next to the *groovy* nodes to absorb part of the tension on their buffers. In other situations, the ring might have to shrink and the number of queries picked by each node for execution dynamically adjusted. The trade offs of this dynamic adaption are presented in detail in the coming Chapter 6.

5.5 Application layer evaluation

The the Data Cyclotron is designed for applications with complex analytic queries, I/O intensive and high throughput demands. The following study demonstrates how a MonetDB should exploit its features to achieve high throughput with low query response time. It also identifies some boundaries of the architecture imposed by the hardware configuration and how they could be solved when a system is assembled from scratch.

For the experiments TPC-H is used as a frame of reference. TPC-H is an important decision support workload supplied by the Transaction Processing Council [19]. It contains 22 queries which analyze relational tables for decision support for commercial enterprises. Disk I/O is typically the limiting factor in the query performance of TPC-H. The I/O incurred is due to reads rather than write operations. A particularly characteristic for decision-support workloads.

The queries complexity and the different data access patterns create a challenging environment to identify design issues on the early stages development of a novel architecture such as DaCyDB.

5.5.1 Access to remote memory

In workloads like TPC-H, disk operations are so dominating that the CPU utilization is below the 50% utilization. Waiting for disk operations to complete, not only results in poor CPU utilization, but it also significantly increases the system call overhead. Hence, the Data Cyclotron exploits access to remote memory to reduce as much as

[19]http://www.tpc.org/

Query	1 node	32 nodes	48 nodes	64 nodes
1	208.4	11.5	14.1	14.6
2	13.6	3.6	4.9	4.8
3	14.7	8.3	8.6	10.9
4	30.4	4.8	5.4	5.8
5	16.9	7	8.6	10.1
6	3.4	5.8	5.3	6.3
7	7.6	10.6	13.7	17.6
8	20.6	10	9.9	11.1
11	11.3	3.9	5.9	4.6
12	15.2	5.6	7.4	7.7
14	4.9	5.1	5.6	6.3
15	13.9	12	12.8	12.6
17	36.4	37.9	73.4	74.9
18	125.2	78.9	108.1	112.8
19	79.3	22.6	28.3	28.8
20	11	7.6	7.7	8.2
21	31.8	10.2	15.8	16.5
22	14.9	8.11	9.6	10.2

Table 5.6: Query execution times.

possible the access to the local disk. It claims, with current network hardware, that access to remote memory is faster than access to a local disk.

In the same line as the micro-bench mark experiments, i.e., ring size and hot-set size, we show how access to remote memory outperforms a single node disk access. For this experiment we have used *Cluster 1* and the data is distributed uniformly among all nodes. Using TPC-H scale factor 40 and executing all queries in sequence at one of the nodes, the response time boost is evident. The query execution time improvement stems from the distributed data load using all nodes. Complex queries that require a scan of the complete lineitem table, like $Q1$, $Q5$ and $Q21$, profit most from the parallel data load as represented on Table 5.6.

For ring $R1$, there are 32 nodes reading data concurrently. It behaves like a networked storage system with 32 disks with access bandwidth near to 1.3 GB/sec on *Cluster 1*, not far from the 2 GB/s bandwidth achievable by an expensive state of the art RAID system with 4 SSD disks. However, if such a system is not integrated with DMA channels, CPU cycles will be shared between data transfer and data processing. With the Data Cyclotron, and as a result of the RDMA benefits (cf., Section 2.6.2), the CPU was fully devoted to data processing. Therefore, remote data access is overlapping with data processing, which explains the constant gain for $R2$ and $R3$.

The results in Table 5.6 can be divided into two groups, network I/O bounded

#Nodes	Queries/sec
1	3
32	5
48	7
64	9

Table 5.7: Throughput.

queries and disk I/O bounded queries. By network I/O queries we classify queries for which the access to remote memory increased their response time compared to access to the local disk. In the first group are $Q6$, $Q7$, and $Q14$. With the queries executed one after the other, and 24 GB of available main memory, the three queries benefit from data loaded into memory by previous queries.

$Q5$ loads most of the data needed by $Q6$ and $Q7$. Hence, $Q6$ only needs to load into memory a few columns from lineitem table. Furthermore, due to its low footprint, it does not unload data to virtual memory. With lineitem columns as the biggest ones, $Q7$ is the one that most profits with the access to those columns in memory. Hence, any remote access will be slower than local memory access. The same happens for $Q14$. Two of the requested lineitem columns were in memory from $Q12$ execution. The other three, in worst case scenario, had to be requested which explains why the single node execution time is near to the execution time in a 32 nodes ring.

5.5.2 Distributed cooperative work

With the data flowing trough the distributed ring buffer composed out of remote memories, the queries can be distributed among the nodes to exploit a distributed cooperative work without increasing too much the the query response time. It definitely increases system throughput.

However, it is still a challenge to obtain optimal throughput and the same query response time when more than one node in the ring is executing queries. To study the cooperative work among the nodes we used a workload composed of 1024 queries derived from the TPC-H queries using an uniform distribution. As for the experiments in Section 5.4 each node picks and executes one query at the time.

The results in Table 5.7 follow our intuition, there is throughput improvement. With the hot-set of size similar to $S6$ from Section 5.4.2, the 64 nodes ring shows better response time average for most of the queries. Although, the query response time was not as optimal as expected, see Figure 5.13. A close look at the query trace pointed towards a less than optimal exploit of the continuous data stream. The current

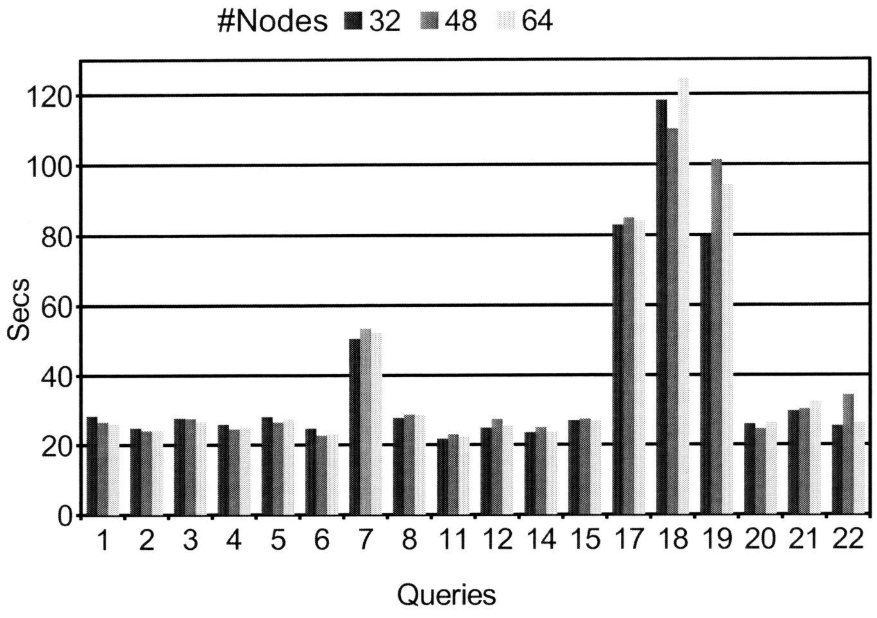


Figure 5.13: Query average time with all nodes.

instruction scheduling algorithm limited each node to work in a distributed cooperative manner.

The current execution model was designed for pure pull-based processing where the plan is static and the data is retrieved as the instructions are executed. The intra-query parallelism through partitioned execution (cf., Section 5.2.3) gave a new degree of flexibility, but by itself is not enough to optimally exploit the continuous stream of data for query plans with high number of dependencies among the instructions and large intermediate results.

Queries such as $Q2$, $Q4$, $Q11$, and $Q19$ are the ones that suffer most from the current query execution model. $Q19$ and $Q11$ stand as the worst cases, their response time increased by a factor five. A close look into the query execution traces, raised a concern already discussed during simulation, but never seen during experimentation. It was observed that most of the data chunks required for processing had just passed by or have recently been evicted from the cache. It exposes the need to integrate a *beforehand* cache policy. The *beforehand* cache policy caches data chunks up front for computations where the *pin()* call was not yet registered due to a dependency, but it will in short-time period (cf., Section 4.5).

" The runtime system through the registration time of the request() *calls, and on previous* pin() *calls, estimates when the data chunk will be used* [20]*. Based on the estimated time, runtime system predicts if a data chunk will be used before it completes another* DaCy-cycle. *If not, and in the presence of enough resources, the data chunk is cached. The data access time is then resumed to a read from the local memory. "*

With this cache policy, the Data Cyclotron reduces the data access latency for data chunks used in less than a *DaCy-cycle*. With those cached, more instructions become eligible for execution, and thus different execution paths on the query plan can be followed. By caching the data chunks passing by, it exploits the actual data stream instead of requesting new data which triggers the un-load and re-load of hot-set data chunks. An important characteristic to have cooperative work between nodes.

This cache policy reduced the data access latency, but it also stressed the virtual memory management. With several instructions from different *Dataflow* blocks becoming eligible for execution, the memory becomes overloaded with the intermediates of unfinished *Dataflows*. The more *Dataflow* blocks are initialized, the higher is the number of intermediates to store. Hence, virtual memory storage is required and due to its low bandwidth, the queries response time increases.

The issue became even more evident on *Cluster 2* which has only 8 GB of main memory for intermediates. After re-running the 1024 TPC-H queries, despite a little throughput improvement due to higher network bandwidth, most queries had a response time increased by a factor two or three.

In both cases, the reason for the high virtual memory utilization is the admission policy used in the scheduler and the static number of workers assigned for query processing. With all workers requesting instructions for execution and a scarce amount of memory to store all intermediates, workers tend to leave the execution of a *Dataflow* block and jump to another one in search of instructions with lower memory footprint. At short term, the decision improves performance. At long runs, it creates a large list of intermediates and the initialization of several *Dataflow* blocks.

As a consequence the scheduler also needs to consider virtual memory load when choosing the right path on the execution graph. The deeper it goes in the execution graph, the instructions input is mainly composed of intermediates. With most of them

[20]The estimation is more accurate if provided by the computation which is possible at registration time.

stored in virtual memory, choosing the path which leads to the lowest virtual memory I/O is a complex task.

Reduce the amount of intermediates.

In the context of TPC-H large intermediates are created. Hence, the number of eligible requests needs to be reduced, to not only to increase the number of available DaCy buffers for input data, but also to reduce the amount of intermediates. Through the configuration variable presented in Section 5.4.4, **X**, the number of eligible requests is reduced.

" *To solve the issue, the* opt_datacyclotron *optimizer was improved to reduce the number of requests eligible for execution. Instead of making all requests eligible, the optimizer makes* **X** *requests eligible while the remain ones become eligible as pin calls are executed.* "

A lower number of eligible requests restricts the number of instruction eligible for execution. Hence, the workers have less *Dataflow* blocks to pick for execution. It forces them to finalize a *Dataflow* block before moving to the next one, i.e., the workers tend to go as deep as possible in the execution graph before moving to the right (cf., Figure 5.4).

Using the new version of *opt_datacyclotron*, we re-run the TPC-H queries with half of the space for intermediate results, i.e., 8 GB. Using the four different values of **X** and with the number of workers fixed to seven. The best query response time achieved was with $X = 20$. Unfortunately, for most of the queries, the reduction of the number of eligible requests was not enough to avoid virtual memory utilization. The reason is the excessive number of workers.

With large intermediates and lack of space to store them, seven workers require the used of virtual memory to store intermediates. Hence, we have re-run the non-blocked queries with less workers. The number of workers was decreased step wise from seven down to three. Figure 5.14 contains the relative gain compared to the runs with seven workers. The improvement on response time is evident.

For four workers the query response time has decreased, in average, by a factor two. For a lower number of workers, the gain starts to decrease. With fewer workers the resources are used inefficiently, such as CPU cycles, and the data access latency increases. With few workers a lower number of requests is registered which contributes to a slimmer hot-set. Hence, each request requires a (re-) load of a data chunk which taxes data access latency.

The queries not represented on the graph are the ones which got blocked in some

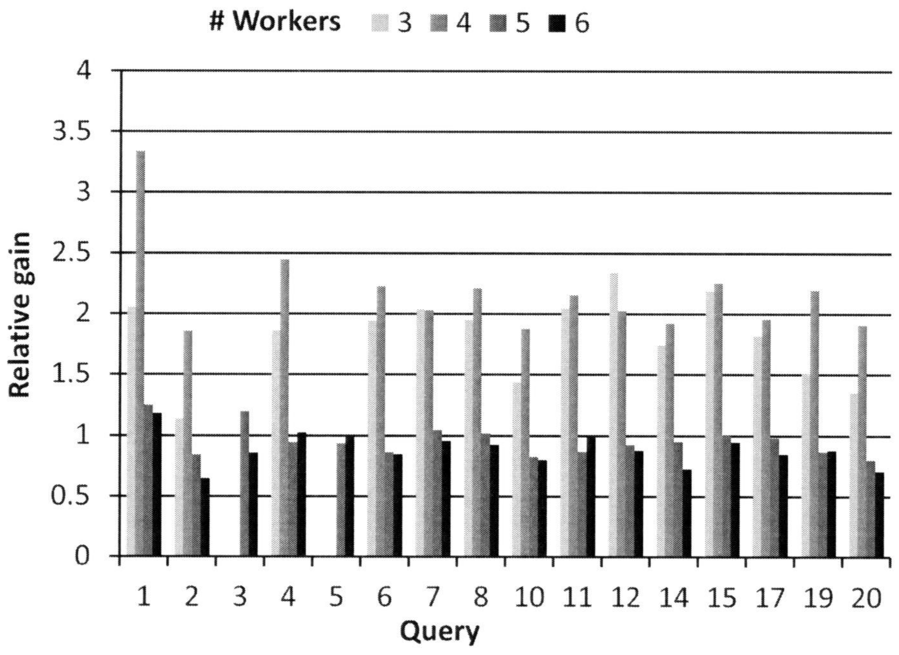


Figure 5.14: Query response time gain compared to seven workers.

of the runs due to high number of dependencies. To solve the issue it required an extension to the *opt_unroll* optimizer to leverage the packing of intermediates, i.e., be more conservative and do not unroll to much the plan. With this generalization of the unroll optimizer it was possible to execute all queries, but with performance degradation on the other queries. The optimal design of this optimizer is part of future research.

The experiment enhances two important features for the design of a new scheduler for DaCyDB. The first feature, the number of eligible requests should be controlled to force the workers to produce small amount of intermediates. Second feature, a dynamic adjustment of the number of workers. For optimal results, the scheduler, based on the intermediates size, should consume the highest amount of intermediates before they are flushed to the virtual memory, i.e., *hot-potatoes*.

Management of *hot-potatoes*.

For $Q3$, most of the virtual memory traffic is created by the *pack()* operators. A trace of their execution shows that the selection of instructions for execution does not consume the *hot-potatoes* early enough to avoid their storage in the virtual memory.

Figure 5.15 is an extraction of the execution plan for $Q3$. Due to resource competition by the workers, the *pack()* input arguments were swapped before the instruction was called for execution. Hence, during its execution, all the BATs have to be read from disk to create BAT X_11. An acceptable cost since there is high competition for resources to execute all *Dataflow* blocks. However, there is an extra cost due to the fact X_11 is not used straight away. Without a direct utilization, and with high competition for resources, the data is once again swapped out. At the time it is needed for execution, the operator has to wait for its read from disk.

The *admission* policy by itself cannot avoid this extra write/read from/to disk. The performance impact by this extra disk I/O can be determine through an isolated execution of $Q3$ with the pack's result being used right away after its creation. Hence, the second write/read to/from disk is avoided. Monitoring the Disk I/O, through the command *iotop*, it was clear that the amount of data fetched in and out of disk has decreased. As a result, the response time for $Q3$ improved by a factor three.

For other queries the packs are used by several instructions or, instructions are dependent of several other packs. For those queries, pulling the pack next to the first operator that uses its result is not enough to reduce the swap problem. Workers thread simply run the hottest and eligible operators first. Furthermore, the scheduler tries to keep all threads busy which leads to an high competition over the available main memory and thus, forcing the use of virtual memory to assure their execution. In addition to the dynamic adjustment of workers, a temporary pause of a worker should also be considered.

The new scheduling model has to be more robust by following the dependency list and for each operator, estimate the intermediate result size, and determine how much swap traffic will be created. Then, define the order which creates the lowest virtual memory utilization. In case of lack of memory, and with the risk to over utilize the virtual memory, a number of workers should be paused. Such dynamic adjustment is complex and imprecision on the decision can limit MonetDB's capacity to scale up. The realization and quantification of such an approach is part of on going research for a different dissertation.

Remote memory as virtual memory?

Without doubts an improved version of instruction scheduling would equip MonetDB with the tools to reduce swapping, but not to evict it for bigger scale factors. For

```
begin query();

... 60 instructions ...

18391    X12 := pack( X1218, X1220, X1222, X1224, X1225, X1227 );

... 13 instructions ...

33953    X1230 := leftjoin( X1147, X389 );
36511    X1155 := leftjoin( X1147, X427 );
3        X1160 := reverse( X1155 );

... 9 instructions ...

11        X117 := request( lineitem, lorderkey, 3 );
30552    X1159 := leftjoin( X1150, X436 );
3        X1163 := reverse( X1159 );

... 2 instructions ...

4        X200 := request( lineitem, lshipdate, 3 );
52767    X1158 := leftjoin( X1149, X433);
3        X1162 := reverse( X1158 );

... 19 instructions ...

48843    X1151 := leftjoin( X1147, X401 );
47561    X1152 := leftjoin( X1148, X402 );
47949    X1157 := leftjoin( X1148, X430 );
2        X1161 := reverse( X1157 );

... 78 instructions ...

562394    X11 := pack( X1160, X1161, X1162, X1163 );
5599118    X13 := join( X12, X11 );

... 146 instructions ...

end query;
```

(a) MAL structure

Figure 5.15: Snipet from Q3 plan execution

example on SF-80, pulling *packs* up was not enough to stop the $Q3$ being bounded by the disk speed. Hence, scale up is not by itself the solution to provide the highest throughput. The integration of a scale out solution is necessary of a flexible architecture and balanced utilization of resources. In any case, the problem can be attenuated with the use of remote swap or faster disks, for example, SSDs. For DaCyDB we have in mind to explore remote memory as virtual memory store since SSDS are only efficient for data reading, not for data writing.

Authors in [115] have taken the advantage of the RDMA and high network bandwidth (InfiniBand DDR) for an efficient virtual memory to sort data. In their experiments they compared the access to data in remote swap, local disk and local memory. In their setting, the local memory was only 1.45 times faster than using high performance networking block device (HPBD), while HPBD was 2.2 times faster than the disk (40 GB ST340014A ATA/ATAPI-6). At the same time HPBD, thanks to RDMA, performed 1.45 times better when using GigE (Gigabit Ethernet), and 1.29 times better than IPoIB (IP over InfiniBand).

In the execution of a quick sort algorithm, application with intensive swapping activity, the HPBD was not 2.2 faster than the disk, but 4.5 times, since reads were not anymore sequential. With network speed approaching what the memory system can deliver, such as InfiniBand QDR, remote swapping with efficient communication protocols emerges as a viable solution to increase performance. The cost of a memcopy is equivalent to a write to a remote memory. In the context of DaCyDB, the virtual memory on local disk would be replaced by remote memory, accessed through RDMA. The virtual memory would mainly be used for intermediate results. They could have been thrown in the storage ring, however, such an approach would slow down the deliver of input data to starving queries.

For an higher scale factor more extra nodes could be added. Instead of pushing only the intermediate results, the instructions that use them as input could also be pushed to exploit the idle CPU of those nodes. In any case, the solution has its limitations due to the high cost of adding extra nodes just for remote swapping and a balance resource utilization. The optimal solution would be the integration of horizontal scalability to have always the highest throughput and balanced resource utilization. In the following Chapter 6, Section 6.2, it is shown how the decentralized structure of the Data Cyclotron offers the means to explore a new scale out solution. A solution that exploits the fact a computation can be divided into sub-computations and they can be executed anywhere in the Data Cyclotron ring.

5.6 Summary

The Chapter introduced DaCyDB, the integration of a MonetDB instance on each Data Cyclotron node. MonetDB was chosen for being a column-store, i.e., column-stores are known for being efficient for intense data analysis, but also for using the operator-at-the-time paradigm and partitioned execution for efficient parallelism. DaCyDB is in favor of partitioned parallelism since it offers much better opportunities for speedup and scaleup then *pipeline* parallelism. By taking the large relational operators and partitioning their inputs and outputs, it is possible to use divide-and-conquer to turn one big job into many independent little ones. This is an ideal situation for speedup and scaleup [46].

The extended plans created for partitioned parallelism are used by DaCyDB to maximize the cooperative actions on the shared data chunks. Such cooperative work is only possible due to the strategy taken by MonetDB to unroll the loops through code repetition (cf., Section 5.2.3) which combined with code re-utilization, such as recycling [84, 83], turns the overhead of defining and executing long plans negligible. This cooperative access to the partitions together with the internal query parallelism contributes for high throughput and low query response time.

Using two different clusters, the Chapter identified the challenges and issues to cope with different bandwidths, traffic jams, un-balanced data distribution, and a flexible query parallelism to exploit a continuous data stream. The conducted experiments presented and studied solutions to make DaCyDB more efficient for complex computations such as queries from TPC-H benchmark. From a better plan optimization to an efficient management of intermediates to reduce virtual memory utilization, several improvements were considered and some tested with experiments. The steps to build a new scheduler, for an effortless and efficient intra and inter query parallelism, were identified. At the same time, the protocols and routines presented for the hot-set management (cf., Chapter 4) were also validated.

The ideas presented and tested during this Chapter define the building blocks of the Data Cyclotron and its integration with a DBMS. The landscape of research is however hardly explored. The following Chapter introduces the most important research direction to build an outstanding architecture for dynamic distributed data analysis.

Chapter 6

Vision for DaCyDB

Despite the changes in the database world, relational DBMSs have tended to dominate the market. They are the default choice for holding the data behind web-applications and scientific applications. A scale-up technology which requires more CPUs, memory and IO capacity when the problem size increases. The price and size of these systems makes them un-attractive for current application scenarios with Petabyte data sets and complex had-doc queries. In response "noSQL" class of databases and MapReduce frameworks have emerged. Solutions such as Hadoop [71], Cassandra [31], Hbase [73], Membase [110], MongoDB [113], and CouchDB [43] are known for their elastic scalability, simpler data models, automatic fault tolerance, and data distribution which leads to lower administration and tuning requirements.

They are designed to scale-out rather than up. They are most appropriate for storing the types of information produced and consumed by web applications, and are optimized for access patterns and consistency versus availability trade-offs that characterize these software systems [108]. In practice, they still require some administration for performance and availability in any mission-critical data store. Most of their feature set is oriented toward the demands of web applications. Nevertheless, data in an application has value to the business and science that goes beyond the insert-read-update-delete cycle of a typical Web application. The increased complexity of data analysis is now requesting also some vertical scaling by those systems. The taken step has been the adoption of techniques and methods used from DBMSs by MapReduce frameworks, such as Clydesdale [89], or the integration with a DBMS such as HadoopDB [2].

The proposed solutions fill a hole in the solution spectrum for distribute data analysis on huge data-sets. A hole created by the decision to move from pure scale up architectures to pure scale out architectures. It is now clear that scale-out and scale-up cannot be thought as two distinct approaches that contradict one another, but rather must be viewed as two complementing paradigms. It does not make sense to go to either of the extreme scenarios, but rather to combine them. The question then becomes, at what point should the services run locally and what time should they be distributed to achieve the scale-out model?

In this Chapter we propose the vision for DaCyDB to answer this question. The DaCyDB detaches itself from those architectures which were only designed to scale out or architectures that were only designed to scale up. It combines both approaches by exploiting the decentralized Data Cyclotron architecture and partitioned parallelism used by MonetDB (cf. Section 5.2.3). Combined with the network road-map it provides a high throughput and flexibility for turbulent workloads with optimal resource utilization.

6.1 Outline

The Chapter is organized as follows. Section 6.2 describes the DaCyDB scale out approach for distributed parallelism. Section 6.3 explains the re-use of intermediates, created by the distributed parallelism, to boost multi-query performance. Distributed parallelism for complex queries is discussed in Section 6.4. Section 6.5 describes how nodes dynamically allocate resources for the query execution, followed by Section 6.6 to propose a multi-ring architecture and fault-tolerance in Section 6.7. The Chapter ends with a summary in Section 6.10 to revive the most relevant features of DaCyDB.

6.2 Scale out approach for DaCyDB

The DaCyDB provides a seamless transition between a scale up solution to a scale out solution by having both in harmony. Based on the resources of each node the computation on-the-fly has the flexibility to explore both models in different degrees. The scale-up model is used for balanced resource utilization and explore data locality using the nodes cache. The scale-out model is used to spread the execution of complex queries across the cluster. With heterogeneous hardware, both models complement each other to achieve high throughput and optimal resource utilization.

The flexibility to move between both extremes and find the optimal niche for efficiency is given by the decentralized architecture of Data Cyclotron and the operator-at-the-time paradigm used by MonetDB. The flexibility comes from the ability to split a query plan into independent sub-query plans and each of them into multiple inde-

```
SELECT c.t_id FROM t, c WHERE c.t_id = t.id;
```

Figure 6.1: SQL statement.

pendent atomic operations. One or more sub-query plans are executed in the same node while others are executed on other nodes. Their allocation depends on the local resources of each node such as number of cores and available main memory for intermediates.

6.2.1 Plan definition

Leaders of the database community have argued [136, 35] that an overwhelming majority of structured data repositories are either star or snowflake schema. Star-schema have proved to be a good way to model large data sets in many industries. Hence, the plan definition to scale out follows a rule of thumb for distributed query parallelism. The dimension data should be replicated while fact data partitions should be spread among the nodes. It exploits data locality for a range of operators such as join. The same approach has been taken in MapReduce context to reduce the amount of data being continuously shuffled [89]. The authors have shown the performance gains for star schema workloads when the rule is applied. However, the data distribution and the query plan optimization were too complex and dependent on how the data is partitioned and assigned to the nodes.

Contrary, DaCyDB decomposes a query plan into a set of sub-functions. The sub-functions are created around the fact columns partitions. DaCyDB identifies the fact columns based on their size or information collect at loading time. The information is used by an optimizer, *opt_split*, to generate each sub-function. The optimizer is called after the *opt_datacyclotron* optimizer to have the necessary calls to interact with the Data Cyclotron.

The optimizer with the list of fact columns takes one fact partition P and builds a sub-function around the instructions that use P as input, or an input derived from P, and all their dependencies until the top of the plan. To exemplify the steps, we use the unroll MAL plan from Section 5.2.3, now pictured in Figure 6.2a. It is the MAL plan for the SQL query represented in Figure 6.1.

Assuming that table t is a fact table and taking the plan in Figure 6.2a, the first function is built for column id partition one ($id.1$). The search starts at the *request()* call and continues until it reaches an instruction which has as input another fact column, or an intermediate derived from a fact column. Such instructions are designated as

```
begin query_Q();
  X2 := request("t","id",1);
  X3 := request("t","id",2);
  X4 := request("t","id",3);
  X5 := request("c","t_id",1);
  X11 := request("c","t_id",2);
  X12 := request("c","t_id",3);
  X17 := pin(X5);
  X18 := reverse(X17);
  unpin(X5);
  X19 := pin(X11);
  X20 := reverse(X19);
  unpin(X11);
  X21 := pin(X12);
  X22 := reverse(X21);
  unpin(X12);
  X9 := pack(X18,X20,X22);
  X24 := pin(X2);
  X27 := join(X24, X9);
  unpin(X2);
  X30 := reverse(X27);
  X33 := leftjoin(X30, X24);
  X25 := pin(X3);
  X28 := join(X25, X9);
  unpin(X3);
  X31 := reverse(X28);
  X34 := leftjoin(X31, X25);
  X26 := pin(X4);
  X29 := join(X26, X9);
  unpin(X4);
  X32 := reverse(X29);
  X35 := leftjoin(X32, X26);
  X15 := pack(X33,X34,X35);          (A)
  X16 := resultSet(1,1,X15);
  X23 := io.stdout();
  exportResult(X23,X16);             (C)
end query;
```

(a) $query\_Q$ before the *opt_split* optimizer.

```
begin query_Q1();
  X2 := request("t","id",1);
  X5 := request("c","t_id",1);
  X11 := request("c","t_id",2);
  X12 := request("c","t_id",3);
  X17 := pin(X5);
  X18 := reverse(X17);
  unpin(X5);
  X19 := pin(X11);
  X20 := reverse(X19);
  unpin(X11);
  X21 := pin(X12);
  X22 := reverse(X21);
  unpin(X12);
  X9 := pack(X18,X20,X22);           (D)
  X24 := pin(X2);
  X27 := join(X24, X9);              (E)
  unpin(X2);
  X30 := reverse(X27);
  X33 := leftjoin(X30, X24);
  dacyExport("query_Q1",X33);        (B)
end query;
```

(b) Sub-computation $query\_Q1$.

```
begin query_Q();
  X33 := resume("query_Q1");
  X34 := resume("query_Q2");
  X35 := resume("query_Q3");
  X15 := pack(X33,X34,X35);          (A)
  X16 := resultSet(1,1,X15);
  X23 := io.stdout();
  exportResult(X23,X16);             (C)
end query;
```

(c) $query\_Q$ after *opt_split* optimizer.

Figure 6.2: MAL plans before and after *opt_split* optimizer.

merge operators. They are often a *join* operator or a *pack* operator. The later packs sub-functions results and is represented as instruction A) in Figure 6.2a.

All instructions, except the *merge* instruction, are packed into a sub-function. The sub-function built for $id.1$ (Figure 6.2b) contains the highlighted instructions from Figure 6.2a and a *dacy_export* result at the end. The *dacy_export* is used to publish the function plan (FP) and the result into the ring as an ordinary data chunk, called *mid-result* [1]. For $query\_Q1$ the result is column $X33$, see instruction B) in Figure 6.2b.

The *merge* operator is then moved into another sub-function ($query\_Q$) proceeded by one or more *resume()* calls depending on the number of dependencies on other sub-functions, see Figure 6.2c. A *resume()* call collects a *mid-result* from the storage ring. It uses FP to identify the *mid-result*. Once picked it resumes the query execution. The injection of *resume* calls interconnects the sub-functions. Any of them becomes eligible for execution once the respective *mid-result* becomes available in a local buffer.

The process is repeated for all fact columns partitions until the optimizer reaches the instructions to export the final result to the user. In this example, two more sub-functions are created, one for $id.2$ and another for $id.3$ [2]. With only two levels of sub-functions, *sql.exportResult()*, instruction C) in Figure 6.2b is also added to sub-function $query\_Q$, Figure 6.2c. Hence, $query\_Q$ is the sub-function responsible to merge all results from sub-functions $query\_Q1$,
$query\_Q2$, and $query\_Q3$, and return the final result to the user.

Each sub-function is posted on the ring for execution and the export of the final result is executed at the local node. Depending on the number of slots ate each node, one or more sub-functions are picked for execution by other nodes. Once picked, the *resume* instructions, together with the *request* instructions, are the first ones to be executed.

At the first glance, with the query broken into sub-functions, the plan seems to create some data redundancy and extra load in the storage ring. However, since the plan distribution is based on the fact partition, the largest partition, only one node requests its load which combined with caching policies, explained in Section 4.5, the large data chunk is quickly removed from circulation.

Furthermore, only the dimension data, which are orders of magnitude smaller, and *mid-results*, which are often a result of a selection, a filter or a join, are flowing around. Similar traffic is created in MapReduce frameworks through distributed cache [70] to replicate the dimension data (cf., Section 3.6).

" Using a distributed file system has also another drawback. Sometimes all or many of the tasks in a MapReduce job list need to access a single file or a set of files. For example, when joining a large file with a small file the predominant approach is

[1]The name mid-result is used to distinguish this type of intermediate results from the ones produced within a sub-function execution

[2]Since their plans are very similar, their MAL plans are pictured in a later Section 6.3.3 for another discussion. In case of interest, Figure 6.8a and Figure 6.8b

to open the small file as a side file. The small file is opened directly in the map task rather than be specified as an input to the MapReduce job. It is opened, loaded into memory and used for the join in the map phase [59].... When thousands of map or reduce tasks attempt to open the same HDFS file simultaneously a large strain occurs on the NameNode *and the* DataNodes *storing that file [59]. To avoid this situation MapReduce provides a distributed cache [70]. The distributed cache allows users to specify any HDFS file they want every task to have access to. These files are copied into the local disk of the task nodes. The approach adds extra overhead before the job can be initialized and the data might need to be re-shuffled again when different jobs are submitted. "*

The overhead imposed by distributed cache does not exist in DaCyDB. The dimension data is loaded into the hot-set as sub-functions are requesting it. Once loaded it is shared among all nodes, therefore, there is no need to re-shuffled the dimension data in case new jobs are assigned to different nodes.

In Section 6.3.3 we proposed a solution to reduce the load of dimension data by packing the common instruction between each sub-function into a single sub-function and re-use its output to avoid the re-load of all dimension data for future computations.

6.2.2 Dynamic plans

The sub-functions, interconnected through the *resume()* instructions, form a logical computation graph that is automatically mapped onto physical resources at runtime. There may be many more vertices in the graph than execution cores in the computing cluster. Such difference has been used in the MapReduce world to create waves, i.e., more map tasks than map slots. Having more sub-functions than the number of slots helps to balance the workload across the cluster.

The resources available at execution time are not generally known at the query definition time. With heterogeneous clusters, the node where the query is compiled into a set of sub-functions might have less resources, such as available main-memory, than the nodes where most of the sub-functions are executed. In this situation, the plan unroll before the sub-functions definition tends to be more conservative. It leads to inefficient resource utilization and constraining vertical scalability.

On the other hand, if the sub-functions were defined in a wealthy node, their execution will request nodes with high number of available cores to quickly process the instructions and consume the produced intermediates to not overload of the main memory with intermediates and consequently their split to the virtual-memory.

Hence, after being picked for execution, a sub-function can be unrolled or rolled up based on the local available resources. For unrolling, the rules are the same as the ones

```
begin query_Q1'();
  X2 := request("t","id",1);
  X5 := request("c","t_id",1);
  X11 := request("c","t_id",2);
  X12 := request("c","t_id",3);
  X24 := pin(X2);

  X17 := pin(X5);
  X18 := reverse(X17);
  unpin(X5);
  X27 := join(X24, X18);
  X30 := reverse(X27);
  X35 := leftjoin(X30, X24);

  X19 := pin(X11);
  X20 := reverse(X19);
  unpin(X11);
  X28 := join(X24, X20);
  X31 := reverse(X28);
  X36 := leftjoin(X31, X24);

  X21 := pin(X12);
  X22 := reverse(X21);
  unpin(X12);
  X29 := join(X24, X18);
  X32 := reverse(X29);
  X37 := leftjoin(X32, X24);

  unpin(X2);
  X33 := pack(X35,X36,X37);
  dacyExport("query_Q1'",X33);
end query;
```

Figure 6.3: Unfold sub-computation $query\_Q1$.

used in Section 5.2.3. For example, $query\_Q1'$ in Figure 6.3 is the result of unrolling $query\_Q1$ from Figure 6.2b. Instead of packing and do a single join (instruction D and E in Figure 6.2b), the join is unrolled and the packing only occurs at the end of the sub-function. The end result is represented by independent instruction blocks in Figure 6.3.

The DaCyDB tries to extract parallelism within a query by exploiting the fact that dependencies are all explicitly encoded in the flow graph. Depending on the data distribution and workload, the number of partitions for the fact data and dimension data can be increased using dynamic slicing. Hence, through dynamic partitioning inter- and intra- sub-function parallelism is increased even further. Horizontal partitioning on the fact data increases the number of independent sub-functions, while on dimension data increases the number of independent instructions within the sub-functions.

With dynamic plans DaCyDB has balanced resource utilization, but it also solves a resource allocation problem common in distributed processing, data skewness. For example, a sub-function B is not eligible for execution until all its input arguments have been produced. Hence, if one of the inputs is from a sub-function A for which the execution is been slowed down due to data skewness, A is decomposed into a set of sub-functions.

The set of sub-functions is sent to ring to be picked up for processing. The merge of their outputs is then passed to B. In case the merge result is too big, the same approach can be applied to B. With the data chunk in memory, and with RDMA, slicing is a cheap procedure. To materialize it, the memory region which defines the view is sent to the storage ring to be picked for execution or stored as persistent data.

The data once re-organized is used by other queries to boost performance. The same goal has been persecuted by MapReduce frameworks to avoid skewness and have in-memory mappers. Small partitions are defined before job initialization which then results in thousand of mappers being scheduled [72]. The static definition at the job initialization time combined with their static data partition and data distribution does not always improve performance due to un-optimal resource utilization.

6.2.3 DaCyDB user interaction

The user expresses his/her computation through *Jason*, *SQL*, *Xquery*, *SparQL*, and *SciQL*. The computation is then compiled into a MAL plan. Using the procedural language from the middle layer of MonetDB, DaCyDB keeps the definition of plans and their dynamic adjustment for efficient parallelism transparent for the user.

A solution that exploits the split between the logical schema and the storage schema. It allows the user to use high level languages to express the computation and the same time access heterogeneous data sources to be processed in heterogeneous hardware. A goal persecuted since the downs of the DBMS, (cf., Section 1.3).

" In the earliest DBMS, beyond efficiency, the aim was to make the data independent of the logic of application programs, i.e., the split of the logical schema from the storage schema, so the same data could be made available to different applications. "

With this clear split between the schemes and the possibility to *crumble* MAL plans several times into independent sub-functions (cf., Section 6.2.2), the flexibility to scale and explore all available resources makes DaCyDB to stand out from MapReduce solutions. They embed computations in a scripting language in order to execute programs that require more than one reduction or sorting stage. In the past years, new query processing systems were built on top of Hadoop providing different languages. The most well known are Jaql [64], Pig [123], and Hive [138]. They are used to express complex computations on the different stages of data processing.

Pig [123] and Jaql [64] are high-level languages for composing and executing complex dataflows on Hadoop. Pig and Jaql focus on supporting ETL-like workflows that require complex data transformations. Like Hive, they also support sort-merge and hash join strategies, and are also constrained to joining two tables at same time.

Pig.

Authors in [60] have proposed Pig. It is a dataflow system that aims at a sweet point between SQL and MapReduce. It offers SQL-style high-level data manipulation constructs which can be assembled in an explicit dataflow and interleaved with custom Map- and Reduce-style functions or executables. Pig programs encode explicit dataflow graphs as opposed to implicit dataflow in SQL.

Pig dataflows can interleave built-in relational-style operations like filter and join, with user-provided executables that perform custom processing. A schema for the relational-style operations can be supplied at the last minute. It is convenient when working with temporary data for which system-managed metadata is more of a burden than a benefit [60]. For data used exclusively in non-relational operations the schema does not need to be described at all [123].

Pig compiles these dataflow programs, which are written in a language called *Pig Latin* [117], into sets of Hadoop MapReduce jobs, and coordinates their execution. The steps to create the Hadoop MapReduce jobs are pictured in Figure 6.4. From the Pig Latin script a logical plan is created with the dependencies. Out of the logical plan a physical plan is created based on the data partitions and their location. Once created, the physical plan is broken into map- and reduce-task. The synchronization points of a physical plan are the delimiters of each task. For example, a *Global Rearrange* requires a reduce task.

Hive.

Hive [139] provides a relational model for Hadoop and a SQL interface called HiveQL. Hive's tables are analogous to tables in relational databases. Each table has a corresponding HDFS directory, but it also supports external tables defined on data stored, NFS or local directories. Users can load data from external sources and insert query results into Hive tables via the load and insert data manipulation (DML) statements respectively. Multi-table insert is also supported.

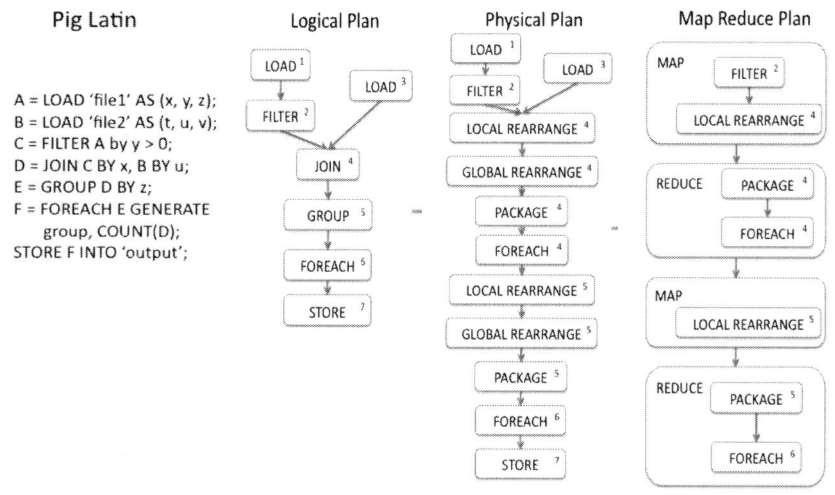


Figure 6.4: Steps to create the Hadoop MapReduce jobs in Pig [60].

Through HiveQL, the user can use statements like *select*, *project*, *join*, *aggregate*, *union all*, and sub-queries in the *from clause*. Multiple queries can be performed on the same input data using a single HiveQL statement. Hive optimizes these queries by sharing the scan of the input data. It exploits cooperative work as Data Cyclotron.

Hive turns HiveQL queries into a series of independent MapReduce jobs to be executed on a Hadoop cluster. The initialization of each job and checkpoints storage between them has a considerable overhead which taxes the query response time. Solutions have emerged to reduce the overhead. The study in [116] focused on grouping MapReduce jobs that perform common computations and evaluating each group as a single job. In *MapReduce Online* [40] the authors modified the Hadoop architecture to allow pipeline of the intermediate data between operators. For example, the reduce output from job A can be used as input for the mapper job B, this is, incremental input pulling.

Pig and Hive are the first step towards to the complete split between the logical schema and storage schema in MapReduce frameworks. There are also prototypes, such as FlumeJava [32], with the same intention, but not so successful as Pig and Hive. Nevertheless, the user is confined to a single language, Latin Pig or HiveQL, to define computations. For complex queries, since they do not support all relational operators, they still rely on user-defined functions written in Java. They are seen as black boxes during query plan optimization.

6.2.4 DaCyDB data access and distribution

Depending on the type of the source nodes, the Data Cyclotron provides two data distribution models: *iterative loading* and *a priory loading* (cf., Section 3.6).

" For workloads which request data from a data source which is always online and has a stable connection with one or more nodes in the ring, the iterative mode should be the preferred one. Otherwise, all data should be loaded before any computation starts. For both, a schema is first shared among all nodes. A request is defined based on this schema and might have as target different data sources. It is composed of the source file identification, parse expression, and split function. For example, a request to extract a relational column from a CSV file, or to extract records from a data block stored in a cloud storage file. "

The latter is the used one in most of the architectures for distributed processing. For DaCyDB, thanks to the Data Cyclotron flexibility and the MonetDB storage model, both are supported. It leads to an effortless data partition/distribution scheme and efficient data access.

For data partitioning and schema distribution, the Data Cyclotron does not impose any requirement other than that each data chunk fits into a single DaCy buffer. Hence, the database is freely partitioned vertically and horizontally using any database partitioning technique to increase locality of access.

Iterative loading for relational schema.

With the flexibility to partition in both dimensions, the *iterative loading* in the context of relation databases is achievable through a small extension to the *COPY FROM* SQL statement and an a priory load of the relational schema.

Preceding the first query execution, a user needs to specify the schema and how the tables should be populated. For *iterative loading*, the loading statements are only used to identify the columns and how to populate them, but not to load the data. Such loading information can always be updated by executing a new $COPYFROM$ statement.

```
a) CREATE TABLE T1 (a int, b int, c string, d double);

b) COPY INTO T1(a,b,c) from 'path/t1.csv'
           USING DELIMITERS '|', '| \n' TOKENS (0,3,20);

c) COPY INTO T1(d) from 'path/time.csv'
         USING DELIMITERS '|', '| \n' TOKENS (4);

d) SELECT a, d FROM T1 WHERE d > 10.0 LIMIT 100;
```

Figure 6.5: SQL statements to populate table **T1**.

Figure 6.5 has the statements to populate a four column table with data from two CSV files [3]. Statement $a)$ can be executed at any node and it leads to the creation of table $T1$ on the local schema. The statement is then propagated counter clockwise for $T1$ be added to all node's schema. For each column an entry is added to the DaCy catalog and marked as unloaded.

Statements $b)$ and $c)$ provide the information on how column **a**, **b**, **c**, and **d** should be populated. With $b)$, column **a**, **b**, and **c** is populated with data from the first, fourth, and twenty first property from $t1.csv$. With $c)$, column **d** is populated with data from a different file, the fourth property from $time.csv$. The execution of $b)$ and $c)$ does not load any data into the DaCyDB. It only adds the loading information to the DaCy catalog. The load of the data only occurs when the first query is executed such as statement $d)$.

The execution of statement $d)$ triggers the load of column **a** and **d**. During the compilation of $d)$, *opt_datacyclotron* inspects the DaCy catalog and if columns are marked as unloaded, it injects a *load()* call per each input source, so the file is scanned only once. After the *load()* calls, it injects a *eval()* call with the query as argument, see Figure 6.6.

At execution time, the *load()* call triggers the load of requests to load data for each column. A request contains as arguments the path to the input file $Path$, delimiters Ds, tokens ids $Tids$, number of records R, and size of the largest static column type length $Clen$ [4]. All this information was provided by the schema and $COPYFROM$ SQL statements.

[3]Other type of files are also supported.

[4]For table $T1$, column **d** has the largest type, i.e., type *double*, because column **c** will only be composed of OIDs (eight bytes length) to access a string dictionary which is propagated in an independent data chunk.

```
begin query();
  X1 := load(T1, a);
  X2 := load(T1, d);
  X3 := eval(X1, X2, "SELECT a, d FROM T1 WHERE d > 10.0 LIMIT 100;");
end query;
```

Figure 6.6: MAL plan for *iterative loading.*

A request flows counter clockwise and to be picked by a node connected to a *data loader* service (cf., Section 3.6).

" Independently of the data source, the Data Cyclotron provides a service to be installed at the data source called data loader. *A* data loader *establishes a connection with the Data Cyclotron and waits for data load requests. Using different readers, the* data loader *extracts relevant properties or fields, partition them, and send them as data chunks to the ring. "*

At its reception, the *data loader*, using $Clen$ and R estimates how many records it can store in a DaCy buffer [5]. Then it starts reading records from $Path$, it uses Ds to parse each record and extract the tokens identified by $Tids$. At the completion of the first partition for each column, the *data loader*, tags them as *COLUMN_LOAD*, adds the estimate number of partitions, and loads them into the ring.

The partitions do a full cycle updating the DaCy catalogs in all nodes. When the first partition of each of the columns have reached the requesting node, the *load()* call returns and the *eval()* instruction is executed. Its execution evaluates the query by compiling it and sending it through the optimizers pipeline. At this time, the DaCy catalog indicates the columns are loaded, or are being loaded, and based on the number of partitions, the *opt_datacyclotron* injects the necessary calls to interact with DaCy layer and to exploit efficient parallelism.

Heterogeneous data sources.

To have DaCyDB more competitive for data preparation, the information to parse the data source, identify the data structure, and define which records or tokens are relevant, should be provided by the user. The approach is one of the advantages in using

[5]By using the largest column type, the loader makes sure the OIDs inside each partition are aligned. A rule of thumb in column stores to have efficient joins instead of Cartesian products.

```
csvParser( <record_delimiter>, <token_delimiter>,
   <tokens_ids>, <first_record>,
   <last_record>, <sampling_function>
 );

e) COPY INTO T1(a,b,c) from 'path/t1.csv'
         csvParser('| \n', '|', [0,3,20], 0, -1, NULL);
f) COPY INTO T1(a,b,c) from 'path/t1.csv'
         csvParser('| \n', '|', [0,3,20], 500, 1000, NULL);
```

Figure 6.7: A parser for CSV files

MapReduce frameworks, i.e., the user through *inputReaders* specifies how the data should be retrieved and passed to the mappers for processing. With the map function being also defined by the user, the data structure used for processing is defined by the user.

Through a simplification of the $COPY FROM$, DaCyDB asks the users to define a parser as SQL function which calls a *C*, or *C++* user define function (*UDF*). This parser is then used by the data loader to extract data at the source nodes. However, in DaCyDB the output of such functions is always tuples to be loaded into data chunks, this is, column partitions.

For the most common data source types, e.g., *CSV*, *FITS* [6], *NETCDF* [7], and *MSEED* [8], the DaCyDB provides efficient parser implementations. For *CSV* files, DaCyDB has a data parser called *csvParser*. Its signature is presented in Figure 6.7.

Using the record delimiter and token delimiter, it extracts the relevant records and from them, the tokens to be sent to DaCyDB. If a sampling function is not provided, all records between the *first_record* and the *last_record* are considered relevant. Otherwise, the function is used to identify the relevant records between the *first_record* and the *last_record*.

The parser function is loaded into the system through a $COPY FROM$ statement and sent to the data loader once the data is requested. Using the *csvParser* function, statement **d)** in Figure 6.5 is now replaced by statement **e)** in Figure 6.7. In this example, all records are loaded [9].

[6]http://en.wikipedia.org/wiki/FITS

[7]http://www.unidata.ucar.edu/software/netcdf/

[8]http://www.iris.washington.edu/manuals/SEED_appG.htm

[9]To load all records *last_record* is defined as -1.

With this approach, the $COPY FROM$ statement adopts a simpler syntax by replacing the *USING DELIMITER*, *OFFSET*, and *RECORDS* clauses by a SQL function. Such modification on the standard SQL is straightforward and gives a new degree of flexibility and control to the user to load and structure data from heterogeneous data sources. An important feature to drop one of the most important DDBMS drawbacks compared to MapReduce frameworks, this is, flexible data loading.

An intelligent data loader.

The extensions proposed for SQL $COPY FROM$ statement allows the user to define iterative loads. The next step to reduce the cumulative cost in answering the first query is to efficiently extract data from the source nodes. Such efficiency is provided by the data loader. An independent service which its study and conceptualization is out of the content of this dissertation. Nevertheless, we lay down here the most important features to be considered on its conceptualization in the near future.

The optimal data loader is the one that after several iterative data load requests has only scanned the entire data source once. Since in most of the cases the record location is not know a priory, the data loader with each data extraction has to learn about the records location to avoid several scans to the same file. For example, if only few columns are extracted from a *CSV* file, offsets for the remain ones should be saved to extract them quickly in the next request, or even save the remain ones into separate file.

The approach has been taken by NoDB [9]. NoDB for each query retrieves the data directly from the data sources. In each extraction saves the information relative to the location of the records and tuples. Such information is used to do relative jumps in the data source to located other records or tuples, and thus speed the data extraction and reduce the cumulative cost in answering the queries. Over time, a full map over the data set is created and the queries access the data as if it has been loaded a priory into the database.

The approach is novel, however, its fine granularity creates a complex index over the data source. For huge data sets its management becomes unfeasible and too complex. The solution is designed for a central system and not as an independent service for data extraction to feed distributed processing nodes. Nevertheless, DaCyDB and NoDB share some concepts.

The DaCyDB data loader service is similar to a data service provided in cloud environments such as Azure SQL or Azure Blob [10]. They provide efficient stream of data to the application. Azure SQL, allows the application to push some simple selections

[10]http://www.windowsazure.com/en-us/

to filter and specify which data is relevant for loading, and thus reduce network traffic from the cloud to the computational nodes. For Azure Blob, the application through a simple API, i.e., *GETs* and *PUTs*, requests data for processing. A more efficient API has been tested with a prototype. The prototype, with similar characteristics to the DaCyDB data loader, was designed to extract data into column partitions and provide efficient data access to a MapReduce framework [11].

Integrity check.

The *iterative load* reduces the cumulative cost to answer the first query for extreme large data-sets for which only few properties are used for query processing. During the iterative loading, due to the absence of data location, the DaCyDB can also check values allowed in columns and enforce integrity between columns and tables in a distributed manner.

The simplest and most common column are the *NULL* and *NOT NULL*. If not null, type integrity is checked using the *CHECK* constraint on a column. Another common constraint is UNIQUE, which ensures a value in a column is unique within the table which can also be done with a *PRIMARY KEY* constraint.

When a partition enters the ring, the first nodes are responsible to do integrity checks with a single scan of the partition. For the *FOREIGN KEY* constraint, a concept at the heart of a relational database, the validation is bit more complex. The *FOREIGN KEY* relationship with values in another table. Hence, it might trigger the load of extra data into the ring, the *FOREIGN* columns. They have to be all loaded and distributed among the nodes. With the primary key columns distributed among the nodes the entire foreign column(s) is scanned giving a complete turn around the ring. With a single spin around the ring *FOREIGN KEY* validation is completed.

Not all workloads require constraint validation, but for workloads like TPC-H it is crucial for data consistency. The DaCyDB supports integrity rules which are often neglected for distributed query processing due to its high weight in the cumulative cost from the load of the first tuples until the output of the first result. DaCyDB gives the flexibility to the user to define different levels of consistency depending on the type of workload, or search intentions. For example, for a first approach a scientist might just want a raw view over the data to see if there is any interesting correlation, if yes, he can then invest in consistency for more accurate results. A vision shared by the authors of an awarded vision paper [93].

[11]The prototype consisted in an efficient layer to iteratively retrieve data from the cloud service and adaptivaly re-organize it to reduce data access latency. It was designed and implemented by the author of this Dissertation during an internship at Extreme Computing Group from Microsoft Research Redmond. The design was evaluated as a promising solution, which validates the research directions for DaCyDB.

6.3 Result caching

The *mid-results* can be cached and re-used by other queries/jobs to avoid (part of) the processing cost, and thus boost performance. They are simply treated as persistent data and pushed into the storage ring for interested queries. Combined with the intra-query parallelism, multiple sub-functions originated from a single query in execution create a large flow of *mid-results* to boost others.

Like base data, *mid-results* are characterized by their age and their popularity on the ring. They only keep flowing as long as there is interest. To not overload the ring, *mid-results* are kept on the local cache of their creator. If a request is issued, they enter the ring. Otherwise, they are gradually removed from the cache to make room for new ones. The first ones to be evicted are the unpopular ones followed by the ones with low percentage of filtered data, i.e., their size is near the sub-function input size.

The idea was tested for a centralized system, *Recycler* [84, 83]. The *Recycler* speeds up query streams in a self-organizing way by exploiting an operator-at-a-time execution paradigm where complete intermediates are a by-product of every step in the query execution plan. To improve the response time and throughput in the operator-at-time setting, only the management of intermediates cost needs to be kept under control, because the creation cost is always taken by the execution paradigm.

The hypothesis was tested in the context of the operator-at-a-time database system MonetDB, the same DBMS used for DaCyDB. Its abstract machine interpreter was hooked up with a recycler optimizer and run-time module. The optimizer marks operations of interest for harvesting. The run-time support uses this advice to manage a pool of partial results. It avoids re-computation of common sub-queries by extracting readily available results from the pool. To manage the resource pool they defined their policies in three dimensions: instruction matching, investment cost versus savings, and pool administration maintenance [84, 83].

For each instruction to be executed the recycler performs a matching process, i.e., it searches for a possible reusable relational algebra operation in the recycle pool. For each operation executed the recycler decides if it is beneficial to keep the result. Finally, to prevent the pool of intermediates becoming a resource bottleneck itself, operations with low potential for reuse should be cleaned from the pool to reduce the memory usage and the search time. Traditional approaches, such as LRU, and cost-based policies based on plan semantics were used as eviction policies to select instructions for eviction. All policies respect and exploit the semantic relationships among the operations executed.

Despite recycler and result-caching have the same goal, this is, re-use operators/sub-function results to boost multi-query performance, they differ in few aspects. The recycler has finer granularity re-utilization compared to DaCyDB result-caching. The recycler is at operator level while DaCyDB does it at sub-function level. The matching on DaCyDB uses the sub-function plan and its input arguments. The match has to be exact since matching a sub-function plan is more complex than matching operators. Furthermore, using the storage ring as recycler cache provides DaCyDB the opportunity to explore remote memory to keep mid-results even in the presence of workload with high demands of main memory. In recycler it would stress the pool of partial results to be shrunk by dropping instructions with high potential for reuse.

6.3.1 Execution model

At execution time, for a sub-function (C), the DaCyDB sends a request counter clockwise to determine if C was already executed or not. The request contains the sub-function plan and its input arguments identification. If the request returns marked *executed*, or *in-execution*, or in the mean time a matching *mid-result* passes by, the C execution is dropped. Any *sub-function* dependent on C's output will collect the *mid-result* from the storage ring.

The sub-function execution cancellation is only possible for the sub-functions which are not returning the result to the user. The cancellation is achieved through the registration of a request for the *dacy_export* input. At the reception of a matching *mid-result*, the *dacy_export* becomes eligible for execution despite all other instructions have not yet been executed. For example, in $query\_Q2'$ (Figure 6.9) the input for the *dacy_export*, $X35$, can come from instruction F) or from the storage ring. Since it is the last instruction, the query can exit.

The *dacy_export* execution sets the sub-function status to *executed* so no more instructions are picked for execution by other execution threads. The last thread to exit the sub-function execution requests a garbage collection to clean context and intermediates.

6.3.2 Mid-results cache

For workloads with skewed interest and queries with several sub-functions in common, the hot-set is mainly composed of *mid-results*. The number of sub-functions sent for execution is low, therefore, *popular* raw data is not requested for processing. On the other hand, for uniform workloads with high percentage of disjoint sub-functions, popular data chunks and *mid-results* have to compete for the storage ring resources. In this situation, the input and output data (*mid-result*) of a sub-function have high probability to co-exist in the ring at the same time.

Keeping both might improve the performance of some queries, but it might also slow down others since more data has to be forwarded in the ring. On the other hand, removing the *mid-results* from circulation just after they have been re-used reduces the chance to boost performance of future queries with expensive sub-functions due to their computational complexity and/or large input data volumes.

To optimize storage ring utilization and have high global throughput, the *hot-set* management routine needs an extra condition than the $LOIT$ to unload *mid-results* in case the input data to create them is also flowing in the ring. The condition determines the average time to process the computation or re-use the *mid-result*. For determination, the condition considers the LOI of each of the N input data chunks (LOI_i), the size of each of them (CHK_i), the mid-result LOI and its size (MID), the DaCy-cycle [12] duration (DCC), and an estimation of the sub-function's processing time (T), i.e., processing time without data access latency.

$$\text{MIDACC} = LOI \times DCC \tag{6.1}$$

$$\text{INAVG} = \frac{\sum_{i=0}^{N} LOI_i \times (\frac{DCC}{2})}{N} \tag{6.2}$$

$$\text{INACC} = \mathit{INAVG} + \frac{\sum_{i=0}^{N} CHK_i}{\mathit{MID}} \tag{6.3}$$

For the *mid-result* there is not execution time. Hence, the average processing time ($MIDACC$) is, in worst case scenario, the time to retrieve from the storage ring, i.e., a DaCy-cycle. For the computation, if input data is composed of more than one data chunk, the average time to access them is half of a DaCy-cycle (cf., Equation 6.2) because the access to one data chunk partly overlaps with the access to the others. In both cases, the multiplication with LOI (cf., Equation 6.1 and 6.2) gives a weight equal to popularity of each data chunk among all nodes.

For workloads with computations with almost no processing time, i.e., only data access latency, the relative size difference between the input data and the *mid-result* (cf., Equation 6.3) favors the lightest one. Selecting the lightest data chunks more room is left on the storage ring for other data chunks or *mid-results*.

[12] A *DaCy-cycle* is the average time for a package to be sent and return back to the origin (cf., Section 3.4.4).

Algorithm 1 Condition to check if a mid-result should be evicted from the storage ring.

```
if $MIDACC < (INACC + EXEC)$ then
    $drop \leftarrow False$
else
    $drop \leftarrow True$
```

Algorithm 1 checks if the average time to access input data plus processing time is higher than the average time to access the *mid-result*. If yes, the *mid-result* is kept in circulation, otherwise, cached. If no further request triggers its load, the *mid-result* is evicted from memory or stored in the cold-set. The cold-set is only reached if it has been an *in-vogue mid-result*. Once cold down, it is seen as a materialized view which was made persistent. Hence, result-caching, based on the workload, dynamically identifies materialized views which otherwise could only be detected by DBA or static analyzes of the query patterns.

6.3.3 Enhance result caching

Result caching can be used to solve the dimension data redundancy issue when the query plan is split into sub-functions (cf., Section 6.2.1). The sub-functions created have instructions in common. In Figure 6.8 are the other two sub-functions from $query\_Q$ (cf., Section 6.2.1) and the common instructions between sub-function $query\_Q2$ and $query\_Q3$ are highlighted.

Since they are encapsulated into a sub-function, the result of their execution is not re-utilized by others sub-functions. Hence, for $query\_Q$, this highlighted code is executed three times, i.e., for $query\_Q1$, $query\_Q2$, $query\_Q3$, unless they are picked by the same node at different times, i.e., recycler [84, 83] reduces them to a single execution.

Exploiting result caching, the *opt_split()* can be improved to extract these type of instructions into a separate sub-function. The instructions to be extracted are the ones which read dimension data and prepare it through selections and aggregations, before joining it with the fact data. By applying this rule, a new sub-function $query\_Q0$ is created and $query\_Q2$ (cf., Figure 6.8a) is replaced by $query\_Q2'$ (cf., Figure 6.9 [13]).

The resume calls in $query\_Q1'$, $query\_Q2'$, and $query\_Q3'$ make explicit the re-use of $query\_Q0$ result. The mid-result is now available to also be used by other queries. The dimension data preparatory step can be re-used across several queries to

[13]The same happens for $query\_Q1$ and $query\_Q3$.

```
begin query_Q2();
  X3 := request("t","id",2);
  X5 := request("c","t_id",1);
  X11 := request("c","t_id",2);
  X12 := request("c","t_id",3);
  X17 := pin(X5);
  X18 := reverse(X17);
  unpin(X5);
  X19 := pin(X11);
  X20 := reverse(X19);
  unpin(X11);
  X21 := pin(X12);
  X22 := reverse(X21);
  unpin(X12);
  X9 := pack(X18,X20,X22);
  X25 := pin(X3);
  X28 := join(X25, X9);
  unpin(X3);
  X31 := reverse(X28);
  X34 := leftjoin(X31, X25);
  dacyExport("query_Q2",X34);
end query;
```

(a) Sub-computation $query\_Q2$.

```
begin query_Q3();
  X4 := request("t","id",3);
  X5 := request("c","t_id",1);
  X11 := request("c","t_id",2);
  X12 := request("c","t_id",3);
  X17 := pin(X5);
  X18 := reverse(X17);
  unpin(X5);
  X19 := pin(X11);
  X20 := reverse(X19);
  unpin(X11);
  X21 := pin(X12);
  X22 := reverse(X21);
  unpin(X12);
  X9 := pack(X18,X20,X22);
  X26 := pin(X4);
  X29 := join(X26, X9);
  unpin(X4);
  X32 := reverse(X29);
  X35 := leftjoin(X32, X26);
  dacyExport("query_Q3",X35);
end query;
```

(b) Sub-computation $query\_Q3$.

Figure 6.8: MAL plan after *opt_split* optimizer.

improve throughput. Workloads like *TPC-H* style are the ones that can benefit most from this re-utilization.

Another scenario is the SkyServer workload [137, 67]. It contains several queries to first tune search parameters, e.g., brightness and wavelength, and then use them on the search of new objects in space or track their trajectory. By extracting the tuning parameters definition into an independent sub-function, dozens of queries can re-used them and thus improve their response time.

Like the recycler architecture [84, 83], result-caching in DaCyDB is especially suitable for applications with prevailing read-only workload and relatively expensive processing, such as data analytics and decision support. However, the result-caching at the sub-function level instead of the operator level makes DaCyDB more robust to low data volatility. The trade offs and optimization of this solution are however, part of future research.

```
begin query_Q0();
  X5 := request("c","t_id",1);
  X11 := request("c","t_id",2);
  X12 := request("c","t_id",3);
  X17 := pin(X5);
  X18 := reverse(X17);
  unpin(X5);
  X19 := pin(X11);
  X20 := reverse(X19);
  unpin(X11);
  X21 := pin(X12);
  X22 := reverse(X21);
  unpin(X12);
  X9 := pack(X18,X20,X22);
  dacyExport("query_Q0",X9);
end query;
```

(a) Sub-computation $quer\_Q0$.

```
begin query_Q2'();
  X4 := request("t","id",3);
  X26 := pin(X4);
  X9 := resume(Q0);
  X29 := join(X26, X9);
  unpin(X4);
  X32 := reverse(X29);
  X35 := leftjoin(X32, X26);          (F)
  dacyExport("query_Q2'",X35);
end query;
```

(b) Sub-computation $query\_Q2'$.

Figure 6.9: MAL plan to enhance result caching.

6.4 Complex queries

The trends for real-time data mining and business intelligence applications show that queries are becoming more complex and executed on extremely large volumes of data. For these type of applications the division of a query plan into sub-plans for remote execution restricts the application to scale by creating hot-spots with certain operators such as joins. The problem is amplified for data sets which do not fit into the star and snowflake schema, this is, are not organized into fact and dimension columns.

A different query parallelism of finer granularity is required to reduce hot-spots and at the same time exploit large multi-core processors such as GPUs. The dynamic slicing approach presented in Section 6.2.2 is here improved in an attempt to have optimal balanced resource utilization. Instead of spreading the query plan at the sub-function level, queries are spread at the operator level. For example, the sub-function plan ($query\_Q1'$) in Figure 6.3, the join in each block is spread among three nodes. The result is packed at the fourth node. The idea has been explored through a prototype for a distributed join algorithm on a ring, *cyclo-join* [55, 54].

6.4.1 Cyclo-join

The *cyclo-join* explores the effortless and efficient data movement to design a distributed join operator. For ($R \bowtie S$), one relation, say S (partitioned into sub-relations

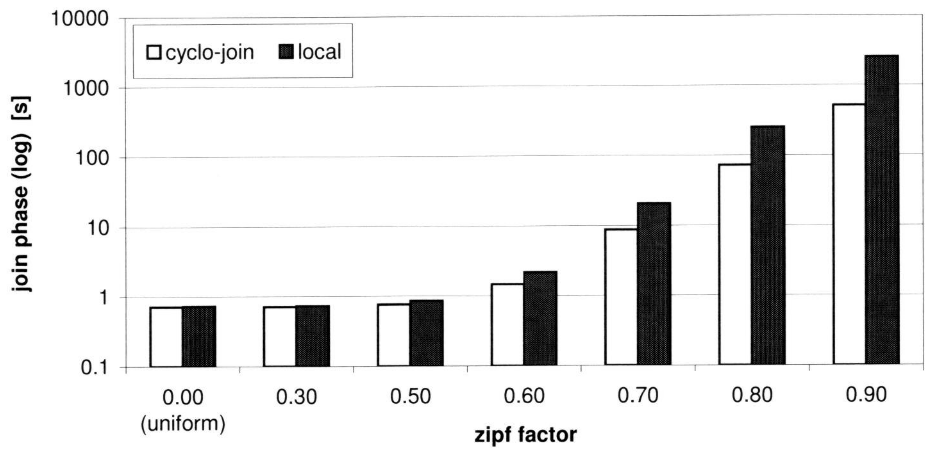


Figure 6.10: Join phase on skewed data [55].

Si) is kept stationary during processing while the fragments of the other relation, say R, are rotating. Hence, all ring members join each fragment of R flowing by against their local piece of S (S_i) *locally* using a commodity in-memory join algorithm. For both joins, the sorted and the hash building is done in a distributed fashion and re-used by other nodes.

The effect of skewed input data on the *cyclo-join* mechanism was studied by generating input tables according to a Zipf distribution with varying Zipf factors z. For various z values input data of size $|R| = |S| = 412\,\text{MB}$ (36 million 12-byte tuples) was generated. For each generated instance the join $R \bowtie S$ was run once on a single host and once on a *cyclo-join* ring that consists of six hosts. Figure 6.10 reports the execution times that we measured for the *join phase* of the partitioned hash join. The setup phase is omitted in this graph since it is unaffected by the data skew.

For Zipf factors of $z = 0.6$ and greater, the exponential increase of the number of *duplicates* in the data sets begins to have a noticeable effect on the execution time of the in-memory hash join. This is not a surprise: the increasing number of *hash collisions* lets hash join slowly degrade toward a nested loops-style evaluation.

The distributed join (white bars) can handle the increasing skew appreciably better. While the processing of uniformly distributed data cannot benefit from a *cyclo-join*-based execution, Figure 6.10 shows a five-fold advantage of *cyclo-join* for input data with a skew of $z = 0.9$.

The benefit comes from two sources. First, the ring buffer mechanism balances differences in the execution speeds of the participating hosts. Thus, a host that is stuck in a chunk of data with a high number of duplicates does not immediately slow down all other nodes. A predecessor only has to start waiting once it has fully consumed all data in its ring buffer.

Secondly, distribution leads to a better use of CPU caches. *Cyclo-join* chops all input data (in particular the inner join relation S) into pieces. Thus, even in the presence of skew, individual partitions within the hash join are less likely to exceed the size of the CPU caches and the join phase can perform more work from within caches.

The concept behind this prototype can be applied to other operators turning a ring of nodes into a powerful DDBMS for complex queries. On one, or more DaCy-cycles, exploiting nodes cache for data locality, a set of independent instructions are executed in parallel. At the first cycle raw data is loaded for processing, such as filtering, and the output directly used as input for another set of operators, such as aggregations, or joined with another raw data. Cycle after cycle, the data passes the different stages that compose a complex query.

With the *DaCy-cycle* as time unit, the execution process can be stretched into a time line with operators overlapping or interconnected in chain representing sequence of filters, aggregations, joins, etc., and the final result being returned in the last DaCy-cycle.

6.4.2 Distributed state machine

The number of cycles is dependent on the number of partitions and the number of nodes which both are dynamically adjusted to achieve the best scalability. Furthermore, an operator can also be executed using several cycles to exploit recursion, such as tail recursion. Tail recursion is a special case of recursion semantically equivalent to the iteration constructors used to represent repetition in programs. Hence, tail-recursive programs can be compiled as efficiently as iterative programs. It turns them more efficient and in addition their definition more clearer. The simplicity of using tail recursion becomes more compelling when the number of instructions and the complexity of their call graph increases.

Hence, a complex query can then be simply compiled into a large finite-state machine with separate tail-recursive functions instead of using a single huge query plan. The parallel execution plan can thus be represented using sophisticated graphs to represent all possible parallelism strategies. For example, DLP-graph (data, precedence and loop dependencies graph).

Contrary to the simple representation of intra-operator parallelism, this is, data dependencies graph (D-graph), a DLP graph models all possible parallel execution strategies by containing the precedence dependencies, such as an operator must be terminated before another operator can start, though no data dependency is involved. Furthermore, they also allow explicitly representation into the parallel execution plan of loop dependency. A loop dependency indicates that a sequence of operators must be repeated as many times as there are available partitions.

With the parallel execution plan defined as a DLP graph, DaCyDB would not be anymore bounded as other systems to static parallelism strategies. It could adopt solutions that integrate run-time control mechanisms. Based on the execution costs, the operators execution could be re-ordered to correct load imbalance, optimizer estimation errors, etc.

As in XPRS [79, 135], a special *choose* operator could be introduced to at runtime to choose different execution alternatives. Another option would be the exchange operator of Volcano. It includes some kind of control, as it can dynamically re-estimate the degree of the intra-operator parallelism [29].

The search space is enormous and old ideas could now be revived for the new hardware trends. The study of the different optimizations as well as the integration of a functional front-end language to define the DLP-graph such as FunSQL [14] [30] is part of the on-going research not covered by this dissertation.

6.5 Dynamic resources allocation

The workload is not stable over time. The search space might have shrunk or the computations became more complex. The immediate consequence is that the initial Data Cyclotron ring size requested by the user may not be optimal in terms of resource utilization and performance. For example, having more nodes than strictly necessary increases the latency in data access while CPU cycles are not exploited. Contrary, having too few nodes reduces throughput. The challenge is to detect such deviations and dynamically adapt the ring structure.

The first step was taken with dynamic plans (cf., Section 6.2.2). They are used to on the fly adjust the query execution plan to achieve efficient intra-node resource utilization. For efficient inter-node resource utilization, DaCyDB uses a *pulsating ring*, one that adaptivaly *grows* and *shrinks* to find the lowest number of nodes comprising a ring to seek the optimal point for throughput.

A *pulsating ring* in a self-organizing way, based on the amount of queries, data flowing and nodes load, they *shrink* or *grow*. It grows to achieve higher throughput,

[14]Extension to SQL which allows application developers to decompose logic into functions (with multiple input and output parameters) and assign functions to bind intermediate results of SQL queries to variables using a static single form (as in functional languages).

but the growth is contended by the data access latency. It does not rely on a central coordinator, all decisions are independently done at each node. It makes the architecture flexible and robust to scale and together with intra-node dynamic resource allocation turns DaCyDB more cost effective than MapReduce- inspired solutions.

Contrary to MapReduce paradigm, the right amount of resources and their efficient utilization is all done by DaCyDB and not through static configuration parameters. The user is thus released from the complexity to specify the accurate amount of resources for complex computations. If not accurate it leads to inefficient resource utilization (cf., Section 2.3.2).

" For complex computations MapReduce solutions looses resource utilization efficiency. The inefficiency is due to three drawbacks, single node resource usage unbalance, reduce slot hoarding, and resource allocation unbalance within a job [142]. ... Last, resources are allocated by a static configuration which does not consider the system dynamical load and job requirements. The number of slots is always fixed independently of the number of mappers or reducers. Furthermore, the cluster might have too many map slots, but not enough reduce slots, and vice-versa. "

6.5.1 Pulsating ring

In DaCyDB computations are not spread based on data assignment, but purely based on node characteristics allowing the system to maximize its resource utilization. They move *anti-clockwise* waiting to be picked up by a node with ample resources. The priority of a computation increases with each transverse of the ring and not be picked for execution. A priority based on the number of cycles works as a timeout for their stay in the ring. Each node selects the queries with highest priority. In case of lack of resources, the node postpones the selection until further release of resources.

When the nodes cannot pick more computations up from the ring it means the nodes are I/O bounded or CPU bounded. In case they are CPU bounded, the ring needs to grow. In case it is I/O bounded, the decision depends on the storage ring load. With low storage ring load, it means the ring is too large, this is, the data access latency is taxed by the large number of hops the data chunk does to reach the destination. With storage ring overloaded, tension of the node's buffers delays the load of data chunks, and thus data access latency is increased.

Any node can request the insertion of an extra node or leave the ring if its resources are not being exploited over a satisfying threshold. In case of an extension, the ring is increased step wise until the data access latency increases the idle time of the CPUs. In case of a contraction, the nodes stop leaving the ring once they start using the *neutral zone* of their DaCy storage for data *in transit*. With homogeneous hot-set, it means

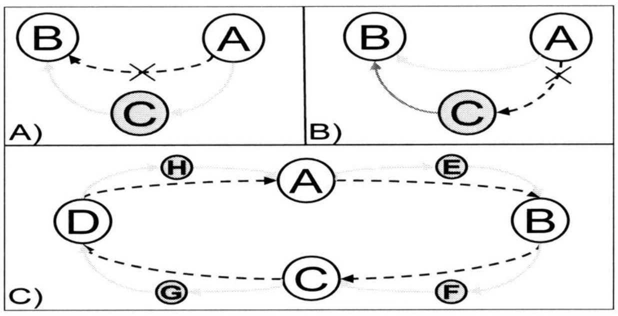


Figure 6.11: Contraction and extension of a ring.

the storage ring is on its optimal utilization and without nodes with tension on their buffers, i.e., there is not *groovy* nodes.

The updates to the ring are localized to the two (envisioned) node's neighbors. A new node simply jumps into the data flow between two nodes as represented in Figure 6.11 A). A leaving node simply notifies its predecessor node and its successor node to make a direct link and to ignore the leaving node from there on. This elastic deformation is bounded by the initial ring size and the number of nodes available for insertion.

More flexibility is obtained if we use the delegate data chunk ownership model presented in Section 5.4.3. On a ring extension a node with high rate of data chunks loads delegates a percentage of its most popular data to the new node. On a ring contraction a leaving node, before disconnecting, delegates the data under its ownership to its neighbors. Since any node can delegate its data, the ring can shrink beyond its initial structure, i.e., it can shrink until the least persistent storage for the data set.

6.5.2 Pulsating ring algorithm

In a pulsating ring the nodes can be added or removed in two ways, one node at the time or several nodes within a single step. In both cases, a lock is used to avoid concurrent structure deformations on the same node. It is assure that no data is lost on the process and it is transparent to all other nodes, i.e., the remain nodes do not notice that a node joined or left the ring.

Lock acquisition.

To leave the ring or to request a new node, a node must obtain the *pulsation's lock*. The *pulsation's lock* is a message that flows around the ring and it contains a status, $locked$ or $unlocked$, and a list with the node's addresses waiting to be added. The node that changes the status from $unlocked$ to $locked$ is the one that has the right to modify the ring structure. Until the status is changed back $unlocked$ no other node can modify the ring structure, i.e., until the lock is released.

For a contraction, it is the left neighbor of the leaving node who releases the lock. It releases the lock after receiving the last chunk from the leaving node. For an extension, it is the new node who releases the lock.

Ring contraction.

A leaving node C, after getting the lock, puts is address in the end of the list, tells its right neighbor A to connect and start forwarding data to B, i.e., C's left neighbor. In mean time C forwards the remaining data chunks on its buffers to B, i.e., B receives data from A and C as shown on Figure 6.11 B). Once C is done with the forwarding, it informs B to shutdown the connection and release the lock. From that point on B only receives data from A.

A node is allowed to leave if its neighbors are not leaving at the same time the ring. Hence, in the extreme case, a ring can shrink one third of its size within a single step. For small rings, to avoid buffer contention due to the double data forwarding (Figure 6.11 B)) only one node at the time is allowed to leave the ring.

Ring extension.

Once a node A gets the lock, it informs the first node on the list, node C, to connect to A and A's left neighbor, node B as pictured in Figure 6.11 A). C initializes its internal structures and routines based on information shared by A. Once ready, it informs A and B to update the TCP sockets or the RDMA context of their transmission/reception threads.

Due to the fact a new node is inserted between the requester and the requester's left neighbor, and the nodes list flows around, the ring extension can be improved to allow multiple extensions one a single step. The *pulsation's lock* type is modified to become a *shared-exclusive* lock, i.e., only one node at the time can leave the ring, but more than one node can request an extension. Hence, race conditions with leaving nodes are avoided and the ring, in the extreme case, can duplicate its size with a single step as pictured in Figure 6.11 C).

6.5.3 Large ring optimization

Workload with large hot-sets tend to request a large ring. The extension to a large ring is normally bounded by the data access latency. The latency to access the less popular chunks and fresh chunks may become a performance hindrance.

As already observed in the Broadcast disks approach [4], the optimal solution to reduce latency in a ring topology ranges from boosting the speed for the most popular data, i.e., increasing their frequency in the broadcast stream, and to cache the highly used ones with lowest frequency. We introduce some changes to the Data Cyclotron architecture to achieve the same goal, although, we take another course.

With the warm-state introduce in Section 4.4, we have partially introduced the concept of seep lines, this is, the data chunk forward is based on a priority instead of *FIFO* order.

" *The Data Cyclotron at load time assumes that all data chunks to be loaded have the same probability to become* standard *or even* in vogue *data chunks. However, not all data chunks are classified as such. In case the loaded chunk is an* unpopular *one low data access latency is assured for few computations, but overall it downgrades throughput. To overcome this issue we propose pre-warming up phase before the data chunk load into the hot-set. In this phase, the data is in the* warm *state, an intermediate state between* cold *and* hot. "

The forwarding thread selects the chunks for forwarding using a Zipfian distribution instead of a FIFO selection. The process is analog, to an highway, where several lines co-exist with different speed limits, i.e., levels of priority. The analogy brings to light the use of shortcuts to reduce the time for a popular chunk to reach a distant node in the ring. It reduces the latency for the most popular chunks. Since the data will only be available at a subset of the nodes, it could lead to throughput degradation, especially for skewed workloads.

To circumvent the problem, we introduce an inner-ring represented as a square in Figure 6.12. The *in vogue* data chunks now flow within the internal ring. Each of the nodes from this ring can fulfill the requests from the nodes ahead of them. For example, node O will fulfill the requests from all nodes between O and C. In case C does not have the chunk it will send a request within the internal ring.

A inner-ring is simplified version of a chordal ring, a simple ring with cross or chordal links between nodes on opposite sides. With different dimensions a chordal ring is used to define several virtual rings to speed up data forwarding of a single hot-set in a large ring. In case the hot-set can be decomposed into disjoint, or partially disjoint, hot-sets a large heterogeneous ring is not anymore the right for efficient re-

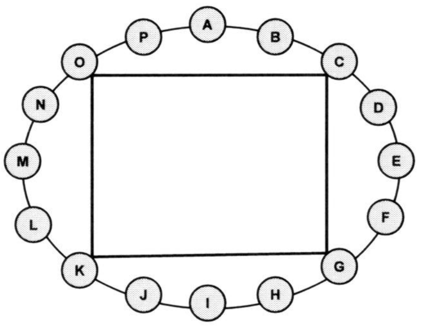


Figure 6.12: Internal ring.

source utilization and low response time. The ring should be fragmented into smaller overlapping, or independent, pulsating rings. The agglomeration of all these rings is called the Data Cyclotron Mesh (cf., Section 6.6).

6.6 The Data Cyclotron mesh

The Data Cyclotron has an innovative architecture with a large scope of scientific challenges. In this section we discuss an even more flexible architecture, the Data Cyclotron Mesh. It is an aggregation of *pulsating rings* Figure 6.13. They are used in the Data Cyclotron to provide an outlook for query processing workloads that can not be addressed with *database sharding* and *map-reduce*. Workloads composed of thousand of queries grouped by common interest on a database sub-set or specific resource requirements.

With the computations crumbled into sub-functions (cf., Section 6.2), DaCyDB exploits the Data Cyclotron mesh to allocate each sub-functions group to a different ring. With a priority policy to pick computations for execution not only based on the age, but also based on the request list and resources requirements, such as virtual memory size or different processing unit, DaCyDB allocates different rings to improve cooperative work, result caching, low data access latency, and in some cases, exploit rings with specialized hardware.

6.6.1 Definition of new rings

The definition of a DaCy mesh starts with an initial ring $R1$. In the initial ring, $R1$, each node observes the request list of each computation passing by in search for access

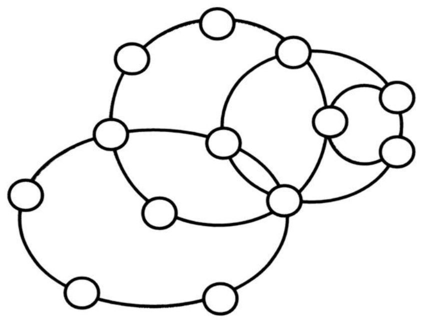


Figure 6.13: The Data Cyclotron mesh.

patterns over the data under its ownership. From the access patterns, a node N derives if its data is only used with a small sub-set of the hot-set, e.g., D_2. If yes, it extracts D_2 from the hot-set and moves it to another ring, R_2. During the process no other node is allowed to build a new ring.

Before the data delegation, N requests a new node to initialize R_2 and, without leaving R_1, creates a two nodes ring. N becomes the *bridge* for the data and computations between the two rings. More than one bridge can exist, if the disjoint set was detected by more than one node. The ring R_2 is then defined by establishing connections between those nodes.

From the access patterns and the computations age, a node is also able to identify a complete new hot-set, D_3. If a set of computations is not collected for execution due to a disjoint list of request compared to the actual hot-set, a node (N) instead of requesting a ring extension, it defines a new independent ring, R_3. It then moves the computations to the new ring. The computations requests will trigger the move of data from R_1's cold-set to R_3's hot-set. In case the data is not present yet in R_1 cold-set, iterative loading is used to load the data from source nodes.

Another example, is the type of resources, such as graphical processing unit (GPU), required by a group of instructions. It is known the GPUs are more suitable for computations involving matrix and vector operations. Using decision models [28] to identify instructions which will benefit from the GPU characteristics, DaCyDB can pack them into independent sub-functions. If a high number of this type of sub-functions, instead of a ring extension, the a new ring, R_4, only composed of nodes equipped with GPUs, can be allocated for more efficient resource utilization.

6.6.2 Inter-ring interactions

With several rings co-existing in the same cluster, computations and data need to be re-directed to the correct ring. To keep the architecture decentralized we add an *outer-ring*. The *outer-ring* is composed of all nodes which were the first node of a ring. These nodes are designated as *primary-bridge* nodes.

Hence, no routing tables or DHTs are required to re-direct traffic between two ring indirectly connected. Once a request cannot be fulfilled in a specific ring, the request is posted in the outer-ring to be picked up by a *primary-bridge* node which has the requested data in its catalog. In case the request returns, the data is requested from the source nodes. The outer-ring is also used to propagate data from one ring to another through ownership delegation or request.

At any point in time, the mesh consists of *numerous* overlapped rings. Each ring is focused on a given subset of the workload. Rings appear, grow/shrink, and disappear as the workload changes. Effectively, system nodes, except the *primary-bridge* nodes, adaptivaly hop from one ring to another to accommodate the workload needs. A ring grows when more resources are needed and shrinks when more than necessary resources are available. Once a ring shrinks completely, i.e., it is only composed of a *primary-bridge* node, it disappears.

The Data Cyclotron Mesh is analog to a crowd in sociology. The continuous adaptation of a ring to the workload demands, without central coordination, behaves the same way as group of persons, i.e., they join an leave the group base on its popularity. If each ring can be seen as a group, the Data Cyclotron Mesh is then seen as crowd organized by groups.

Each group, i.e., each DaCy ring, focus on a specific workload. Over the time, rings become more popular than others, i.e, they grow due to a workload increase. As groups, rings disappear over the time, others split into sub-rings or even merge to form a bigger ring. The interest of the workloads defines the shape and organization of the crowd, i.e., the Data Cyclotron Mesh. The model exploits the independence and autonomy of each individual where everyone, based on the workload flow, works for the same goal, high throughput. As in society, not all individuals are highly satisfied, but a global satisfaction is achieved.

6.7 Fault-Tolerance

In the ring topology the failure of a node interrupts the data flow among the nodes and if a replica does not exist, the data cannot be recovered. To cope with this issue the Data Cyclotron provides two fault-tolerance models: *in-ring* and *out-ring* mode. The fault-tolerance model to be used depends on the type of workload. For workloads requesting data from a data source which is always online and has a stable connection with one or more nodes in the ring, the *out-ring* mode is used. Otherwise, the *in-ring* mode is used.

Data replication.

In both models a data chunk has three replicas. The location of the replicas is what distinguishes a model from the other. For *out-ring* mode, replicas are located at the storage nodes and all of them are used for efficient parallel data access. At the storage nodes data replication is done with multi-primary scheme (called multi-master in the database field). On these nodes, the data is in read only mode. Hence, there is no need for a distributed concurrency control such as distributed lock manager.

For *in-ring* mode, replicas are spread around the ring, i.e., among the node's successors. The data distribution attempts to have two partitions within the same rack and one out-side of the rack. It exploits the fact that nodes from different racks are mixed (cf., Section 5.4.3). It makes DaCyDB robust against a full rack crash. Furthermore, only one replica is set as *primary replica*. The node with the *primary replica* is the one responsible to load its content into the storage ring to fulfill requests.

Connections re-establishment.

For both models the connections are re-established using the Chord lookup protocol [132]. In Chord lookup protocol the nodes keys are arranged in a circle that has at most 2^m nodes [145]. The circle has IDs/keys ranging from 0 to $2^m - 1$. Each node has a successor and a predecessor. The successor to a node (or key) is the next node (key) in the identifier circle in a clockwise direction. The predecessor is counter-clockwise.

Since the successor (or predecessor) node may disappear from the network (because of failure or departure), each node records a whole segment of the circle adjacent to it, i.e. the r nodes preceding it and the r nodes following it. This list results in a high probability that a node is able to correctly locate its successor or predecessor, even if the network in question suffers from a high failure rate.

Data recovery.

With the connection established, the next step is to reload all data lost with the failed nodes and restart their computations. For both modes, when a node fails the requests report back data missing. If the data is not in the hot-set, warm-set, neither cold-set, it will come from storage nodes or from a node in the adjacent list.

For *out-ring* model, using iterative loading, only data in fault is re-loaded from the storage nodes. While loading, the data is accessed by the existent nodes and a new node can be added on-the-fly to become the new owner of the flowing data. With iterative loading, the recovering is an incremental process and it does not affect data traffic on other nodes. Furthermore, there is not replica recovery, i.e., creation of a new copy. The replicas management is done at the storage nodes through traditional replication techniques [74].

For *in-ring mode* data is retrieved from the first node in the adjacent list, i.e, one with a secondary replica. It becomes the *primary replica* and starts to fulfill requests. In mean time a new node, or a successor, is added to the adjacent list. It gets a secondary replica.

The management of secondary replicas is done through heartbeats. The secondary replicas sent heartbeats to the ring to inform the primary replica they are alive. In case one stops sending heartbeats, the primary replica is responsible to re-establish the missing replica.

Computations recovery.

When a computation is picked for execution it does not stop its journey. It hops more three nodes to be replicated. They are call *clones*. In case of failure, the node in the adjacent list used to recover the data, restarts all *clones* it has.

In case the computation has not settled down, i.e., it is not in execution, and the node that holds it on its buffers crashes, the computation cannot be recovered. The same is not applied for sub-computations. They are recovered through time-outs in the *resume()* calls. A *resume()* call contains the sub-computation plan for matching, therefore, when a time-out occurs and the *mid-result* was not retrieved from the storage ring, the sub-computation plan is sent back to the ring for re-execution (cf., Section 6.2.1).

" The dacy_export *is used to publish the function plan (FP) and the result into the ring as an ordinary data chunk, called* mid-result [15]. *For $query\_Q1$ the result is column $X33$, see instruction B) in Figure 6.2b. ... The* merge *operator is then moved into another sub-function ($query\_Q$) proceeded by one or more* resume() *calls depending on the number of dependencies on other sub-functions, see Figure 6.2c. A* resume() *call collects a* mid-result *from the storage ring. It uses FP to identify the* mid-result. *"*

[15]The name mid-result is used to distinguish this type of intermediate results from the ones produced within a sub-function execution

6.8 Green architectures

In the past decades the database and distributed data processing communities have designed and developed a wealth of methods to optimize response time and/or throughput. Energy consumption was however never considered. Economical and environmental factors are requiring to see energy as a critical performance metric for data processing. Energy awareness is opening a wide range of research challenges [100]. New DBMS architectures, such as EcoDB [48], started to consider using *energy consumption* as a first-class metric in a DBMS when planning and processing queries. On the horizon is the definition of energy efficiency metrics for established benchmarks such as those defined by TPC [16] [66].

Nonetheless, the efforts to improve query response time are still very valuable. In most of the situations, faster query execution results in lower energy consumption. However, a more energy efficient data center cannot be realized if there is not energy efficient applications to run on top of energy efficient hardware and networks [36].

From the hardware research community, energy management is already an important metric and design criteria for modern data center management and planning [17]. They are trying to circumvent the problem that the energy consumption of current hardware is not proportional to the actual load on the machine [17, 50].

With DaCyDB we are trying to contribute to this *green* movement. The key feature of DaCyDB, continuous data movement, has raised some concerns about its energy efficiency. With a thoughtful analyze of on going research for energy efficiency, we will show how the DaCyDB architecture has the ingredients to be qualified as a *Green architecture*. Its extensibility for new ideas also grants the conditions to introduce new techniques for energy efficiency.

Efficient resource allocation.

With iterative data distribution (cf., Section 6.2.4), dynamic plans (cf.,Section 6.2.2) and pulsating rings (cf., Section 6.5.1), DaCyDB attempts to process as many computations as possible in a single node and have short rings. For large *hot-sets*, the ring is kept large enough to keep the all hot-set flowing, and thus avoid constant data reloads. A reload often reads data from disks which are known for their energy inefficiency [100].

In case nodes start to be under utilized, they leave the ring to save resources. In case the data access latency increases with the removal of nodes, the ring does not shrink. In this situation, the technique PVC (Processor Voltage/Processing Control) [100] is used to reduce energy consumption. PVC exploits the new state-of-the-art processors.

[16]www.tpc.org

They take commands from software to adjust their voltage/frequencies to operate at different levels than their peak performance. By reducing the frequency of the nodes processors, under-utilized nodes can remain without much energy consumption since all data forwarding is conducted by RNIC using RDMA (cf., Section 5.4) which is more energy efficient than TCP Sockets [104].

The power savings of RDMA are achieved by minimizing the interactions between the network adapters, CPUs and memory. Compared to TCP/IP, RDMA requires much fewer CPU cycles for protocol processing and also generates less memory bus traffic (cf., Section 2.6.2), both of them contribute to its power savings.

" The most apparent benefit of using RDMA is the CPU load reduction thanks to the aforementioned direct data placement (avoid intermediate data copies) and OS bypassing techniques (reduced context switch rate) [13]. ... A second effect is less obvious: RDMA also significantly reduces the memory bus load as the data is directly DMAed to/from its location in main-memory. Therefore, the data crosses the memory bus only once per transfer. The kernel TCP/IP stack on the other hand requires several such crossings. "

DaCyDB also uses PVC for rings with slow networks, i.e., computations are network I/O bounded. Instead of having processing peaks at the nodes, i.e., for each data chunk arrival the processor is taken to its maximum frequency, they process data with lower frequency to align with the data arrival rate without increasing response time.

Topology.

The topology simplicity is another feature of DaCyDB for energy efficiency. With a mesh of pulsating rings the network traffic at the switch is relative simple compared to a $2D$ mesh or partial connected mesh. In DaCyDB the data is transferred in big chunks, constant routing pattern and without bursts. Hence, the DaCyDB traffic requires low synchronization and small amount of resources, such as buffers, to solve contentions (cf., Section 3.2).

" In the Data Cyclotron the data flows clockwise, i.e., it is a continuous stream and with a single routing pattern. Furthermore, with the ring topology, the packets that arrived at different input ports are destined to different output ports, i.e., a contention free scheduling. Therefore, they can be routed instantaneously, i.e, the switches can be non-blocking. "

The data center to host DaCyDB can trade complex switches for simpler switches, and thus the number of watts spend per each GB transfer be reduced. The idea has been explored for on-chip networks. Authors in [26] use a topology similar to Data Cyclotron Mesh (cf., Section 6.6). They use an hierarchical ring topology which consists of small local rings connected via one or more global rings. The design obtains the advantages of both mesh- and ring-base designs, and avoid their shortcomings.

Ring networks use very simple routers, which reduces energy and die area, and eases design and validation, in some situations, performance suffers with scaling (cf., Section 5.4.5). Mesh networks scale relatively well, however, they require large, complex, energy-inefficient routers. Authors [95], for on-chip networks, showed rings achieve very good energy efficiency at low-to-medium core counts, or when network load is low enough that the ring does not become a bottleneck.

The work on on-chip networks emphasizes the fact for scalability energy efficiency, hierarchical ring composed of small rings should be used. When composed of large rings, scalability becomes an issue. DaCyDB uses *inner-ring* to circumvent the problem (cf., Section 6.5.3).

" A inner-ring is simplified version of a chordal ring, a simple ring with cross or chordal links between nodes on opposite sides. With different dimensions a chordal ring is used to define several virtual rings to speed up data forwarding of a single hot-set in a large ring. "

Furthermore, with the *inner-ring* data compression can be used to reduce network traffic, and thus increase energy savings. All data loaded into the inner-ring is compressed. When forwarded to the outer-ring, only the popular data chunks are decompressed to avoid the cost of decompressing data in most of the nodes. The unpopular data is only decompressed at the nodes where it is used, releasing network bandwidth to transfer more data.

Typical compression algorithms, such as gzip, are not the ones to be adopted since they increase the net energy when compression is applied before transmission [16]. Energy-aware lossless data compression [16] used for wireless networks is one of the paths which can be explored since it is an energy-aware solution and efficient as typical compression algorithms. The authors [16] used one compression algorithm on the transmit side and a different algorithm for the receive path. They have shown on their prototype that by choosing the lowest-energy compressor and decompressor, overall energy to send and receive data can be reduced by 11% compared with a well-chosen symmetric pair, or up to 57% over the default symmetric zlib scheme. The choice of right compressor and decompressor for DaCyDB context is part of ongoing research and thus, not cover by the content of this dissertation.

Type of nodes.

Recent studies have focused on redesigning data center server clusters with low cost, low-power *wimpy* nodes [10, 86] because they are relatively well-balanced [127]. These type of nodes are more energy efficient due to their lowend CPUs and low-power components. On the other hand, they lag far behind traditional nodes in performance. Hence, to replace a small cluster of traditional nodes, such as *Xeon* [17] nodes, a larger cluster of low-end nodes is required [101]. This type of clusters are the ones used by MapReduce frameworks which are mainly designed to scale out (cf., Section 2.3.3). However, complex computations exhibit disproportionate scale up characteristics which potentially makes scale-out with low-end nodes an expensive and lower performance solution. Authors in [101] have shown that for data-intensive workloads, a TPC-H workload, large wimpy node clusters suffer from poor scale up effects, and thus are potentially slower and a costlier solution compared to Xeon clusters.

Wimpy clusters are more affected by a diminishing return scale up effect than a smaller traditional cluster [101]. The work shows that in real (as opposed to ideal) scale up environments, price/performance degrades as the scale up factor is increased. It gets more expensive to achieve the same level of performance. Nevertheless, the experiments also showed that using only Xeon nodes was not the direction to be taken. A hybrid solution is the one allows to have low response times and energy consumption [101, 37]. Hybrid cluster deployment strategies, job scheduling, and scale up analysis were mention as interesting avenues of future research [37].

DaCyDB aligns with this line of thinking. It is tailored for complex computations through its seamless transition between the scale up model and the scale out model (cf., Section 6.2).

" The DaCyDB provides a seamless transition between a scale up solution to a scale out solution by having both in harmony. Based on the resources of each node the computation on-the-fly has the flexibility to explore both models in different degrees. The scale-up model is used for balanced resource utilization and explore data locality using the nodes cache. The scale-out model is used to spread the execution of complex queries across the cluster. With heterogeneous hardware, both models complement each other to achieve high throughput and optimal resource utilization. "

Furthermore, with the ring as logical topology, wimpy nodes and nodes like Xeon nodes can co-exist in the same ring. The computations are picked by the nodes based on their resource utilization (cf., Section 5.4.4). In case a computation is too slow, it

[17] http://www.intel.com/p/en_US/embedded/hwsw/hardware/xeon-previous

is decomposed and executed by several nodes through the *dynamic plans* optimization (cf., Section 6.2.2). If not possible the decomposition, the computation is restarted somewhere in a bigger node.

Instead of restarting the computation, the computations status can be frozen and sent for execution in another node, for example, a node with ample CPU cores, a GPU or higher amount of remote memory for virtual memory. Only the intermediates and the remain plan execution are transferred; the data needed will pass by upon request. Hence, in DaCyDB, using compact rings with heterogeneous nodes, complex computations are broken up in pieces for efficient processing and energy awareness.

Query scheduling.

With queries flowing in the storage ring, DaCyDB has the capacity to explore **Q**uery **E**nergy efficiency by introducing explicit *D*elays (QED) [100]. With *QED*, queries are explicitly delayed for workloads in which there are often common components across different queries (and delays can be tolerated). Hence, the queries build up in the queue to be executed as batch. The authors in [100] had energy saves up to 54% using a TPC-H workload on MySQL cluster and commercial DBMS. However, the performance decreased by 43%. The major reason was the inefficiency to exploit the commonalities between the queries. Furthermore, they were merged into a single query and then the result has to be split which is an expensive procedure.

DaCyDB tackles the problem from a different angle. Queries are split into sub-functions and DaCyDB uses them to find commonalities among the queries, i.e., through *result caching* (cf., Section 6.3) DaCyDB reuses sub-functions result avoiding the execution of hundreds of instructions. Furthermore, with sub-functions dependencies defined through a dependency graph, each query result can be built independently without extra cost.

Another energy saving is on disk accesses to retrieve data for the queries. With priority policy to pick queries up for execution, defined for the Data Cyclotron mesh (cf., Section 6.6), the hot-set definition is aligned with the queries batch. Hence, with less shifts on the hot-set, less disk reads have to be performed to load new data into the ring. Disks reads, specially random reads, are known to be energy inefficient [100].

" With the computations crumbled into sub-functions (cf., Section 6.2), DaCyDB exploits the Data Cyclotron mesh to allocate each sub-functions group to a different ring. With a priority policy to pick computations for execution not only based on the age, but also based on the request list and resources requirements, such as virtual memory size or different processing unit, DaCyDB allocates different rings to improve cooperative work, result caching, low data access latency, and in some cases, exploit

rings with specialized hardware. "

6.9 Where does DaCyDB stands compared to other solutions?

The vision for DaCyDB presented an novel and dynamic architecture to coupe with the challenges of current complex workloads for large data-sets while it offers efficient resource utilization on modern hardware. It is not a one-size-fits-all architecture. It is designed to efficient at a particular area in the data processing line. From data collection until the result presentation, several data processing stages are crossed. Hence, for each of them we identify which architectures are more suitable than DaCyDB or when DaCyDB stands as the right system to be used.

In general, the data processing is divided into three stages: data collection, data preparation, and data presentation [58]. The last two are the ones for which the world of data management has payed more attention. Several solutions, such as high performance data warehouses, Hadoop implementations, extreme performance, or data integration technologies, have emerged to cope with the challenges of these two stages. Each solution has its strengths and weaknesses depending on the data processing stage.

In general, the MapReduce frameworks were designed to be efficient on the data preparation stage while DaCyDB was designed for the data presentation stage. Nevertheless, both solutions can also be adapted to somehow be efficient on others data processing stages. For example, Yahoo considers Pig [60] more suitable for the data preparation phase while Hive [139] is more suitable for the data presentation phase [58].

6.9.1 Data preparation stage

The data preparation stage is normally classified into three categories, pipelines, iterative processing, and research [58]. In the first category the pipeline brings in the data feed, such as logs from web-servers, cleans it and transforms it. A MapReduce framework is without doubts the most suitable for it.

In the second category there is usually one very large data set that is maintained through incremental updates. For example, a huge graph where each node is a page and the links the relation between nodes based on the page content. An update resumes to the integration of recent news.

The DaCyDB could fit into the model by having the update data flowing in the ring to update data stored in the cold-set. However, for this scenario latency is more important than throughput which makes the MapReduce paradigm more efficient [45].

In the third category petabytes of data are brought in and researchers quickly write a script to test a theory or gain deeper insight. The data is not always in a nice, standardized state. It is a typical preparation stage on an eScience scenario.

In this scenario the ELT model is the preferred one to load the data, this is, the data is extracted and loaded directly into the data warehouse and the transformation occurs already in the data warehouse. For example, the raw data sets with un-structure data, e.g., EBAY logs, are data-sets that cannot be a priory structured for further analyzes. All relevant information is extracted, combined and analyzed at runtime.

The process requires a full scan of the entire database which is suitable for MapReduce. However, the DaCyDB architecture stands here as promising architecture to compete with MapReduce frameworks. The elements of a ring would scan the unstructured data and turn it into semi-structure data for analyses. Breaking up the web-click records in essentially key-value sets which are stored in binary columns floating around the DaCyDB ring.

The ring can be composed of nodes with different functionalities. Given the inherent cost of parsing, several may be chartered to read and tokenize the data, while others could be chartered to hunt for specific patterns or perform a learning algorithm. The data needed remains available in the ring until all parties interested have seen it, where after it can be garbage collected or stored for later use.

With iterative load and the possibility to apply different levels of integrity (cf., Section 6.2.4), sub-sets are loaded into the ring for inspection and be correlated with other data sources. The architecture allows an incremental database exploration, where the amount of processing can be controlled by the priority and size of the individual rings.

6.9.2 Data presentation stage

The data presentation stage is classified into two dominant use cases, business-intelligence analysis and ad-hoc queries [58]. In the first case, users connect the data to business intelligence (BI) tools to generate reports or do further analyses. In the second case, users run ad-hoc queries issued by data analysts, decision makers, or scientists.

In both cases the ETL model is often the one used to load the data, this is, the data is extracted to a staging database, transformed in the staging database, and loaded to the data warehouse. The MapReduce is often used to transform and load the data. Furthermore, its explicit and scalable dataflow paradigm has become popular for this applications in favored the traditional high-level declarative approach SQL due to its simplicity.

However, the extreme simplicity leads too much low level hacking to deal with the many-step, branching dataflows that arise for complex computations and standard operations such as joins which are repeatedly code by hand [60]. Pig and Hive are the best examples to move the user away from this low level hacking and complex computations optimization. However, they are limited by the initial design of MapReduce for data distribution (cf., Section 3.6), complex computations (cf.,Section 2.3.4, and resources utilization (cf., Section 2.3.2).

Furthermore, the relational model and SQL are the best fit. "Indeed, data warehousing has been one of the core use cases for SQL through much of its history. It has the right constructs to support the types of queries and tools that analysts want to use. And it is already in use by both the tools and users in the field" [58]. DaCyDB through the relation operators from MonetDB and the flexible architecture of Data Cyclotron stands here as a robust and efficient solution for the challenges posed by the data presentation stage in the eScience world.

6.10 Summary

The DaCyDB realizes the idea of data movement between network nodes as an ally for improved system performance, flexibility, and query throughput. The absence of static data allocation is exploited to explore different algorithms for distributed processing. Computations are split into sub-computations and sent to the ring. Each of them settles on a different node following the basic procedures of a normal computation. They are processed concurrently and the individual *mid-results* are then combined to form the final computation result. Such freedom gives the grounds for an application to scale out without be bounded to any complex scheduling algorithm or data distribution scheme.

Such flexibility combined with the elegant operator-at-the-time paradigm used by MonetDB opens doors to explore query parallelism with finer granularity such as at the operator level. With cyclo-join it was shown a promising future for new distributed operator algorithms to exploit the new network trends and the new multi-core processors. Furthermore, combined with iterative data loading and distributed consistency check, the the cumulative cost to answer the first queries is significantly reduced.

Dynamic resource allocation is achieved due to the fact a node is not assigned to any specific responsibility other than to manage hot data in its memory buffers and cold data on its attached disks. Hence, the queries are not tied to be executed to any specific node or group of nodes. Instead, each query searches a lightly loaded node to execute on; the data needed will pass by upon request. This way, the load is not spread based on data assignment, but purely on the node's characteristics and on the storage ring load characteristics. This innovative and simple strategy intends to avoid hot spots that result from errors in the data allocation and query plan algorithms. Furthermore, each node, based on the local resources utilization, decides to leave the ring or request a ring extension. This autonomous and dynamic adaption to accommodate the workload requirements is what defines *pulsating rings*.

The pulsating rings used in the Data Cyclotron provide an outlook for query processing workloads that can not be addressed with database sharding and map-reduce. It is a new concept to design fully flexible distributed query processing architectures. Moreover, it is conceivable that multiple pulsating rings live in the same physical cluster. At any point in time such a mesh consists of numerous overlapped pulsating rings. The model exploits the independence and autonomy of each individual where everyone, based on the workload flow, works for the same goal, high throughput.

The Data Cyclotron opens a vista on a research landscape of novel ways to implement distributed query processing. Cross fertilization from distributed systems, hardware trends, and analytic modeling in ring-structured services seems prudent. Likewise, the query execution strategies, the algorithms underpinning the relational operators, the query optimization strategies and updates all require a thorough re-evaluation in this context.

Chapter 7

Concluding remarks

This Dissertation presents the rise of a new approach for distributed data manipulation exploiting the new network hardware trends. It opens a vista on a research landscape of novel ways to implement distributed query processing. The research presented is a response to the call-to-arms in [92], which challenges the research community to explore novel architectures for distributed database processing.

In answer to such a call, the Data Cyclotron architecture was designed. It addresses the grand challenge of distributed query processing, come up with a self-organizing architecture which exploits all resources to manage the hot data set, minimize query response time, and maximize throughput without global co-ordination.

With a turbulent data movement through a storage ring built from distributed main memory and capitalizing the functionality offered by modern remote-DMA network facilities, the Data Cyclotron uses data movement between network nodes as an ally to improve system performance, flexibility, and query throughput.

The conceptualization of a full functional system was done with an harmonious integration of the Data Cyclotron with a DBMS, more precisely, a column-store. It is an outstanding solution for distributed data analysis on huge data warehouses. From multi-query parallelization to scalable processing of complex queries and from pulsating rings for dynamic ring size adjustments to the co-existence of multi-rings within a single cluster, the research paths to be explored are enormous. Therefore, in this Chapter we summarize the major contributions of this dissertation and the future research steps.

7.1 Contribution

With the recent developments in the research world to enforce new radical changes on the design of network topologies, the Data Cyclotron stands as one of few architectures without central coordination that has an elegant way to distribute and access data. At the same time, it has the grounds of flexibility to explore new and novel data analyze techniques using state-of-the-art hardware.

The Data Cyclotron has adopted the ring topology for a decentralized architecture where nodes share data or parallelize data computations without a central coordinator. Its simplicity is the key to explore new and un-orthodox algorithms for distributed parallel processing. Its communication pattern leverages the data routing latency and gets optimal bandwidth utilization at networks switches using simplified routing.

For a flexible integration with different applications, each Data Cyclotron node is defined by three layers, the network layer, DaCy layer, and application layer. Their interaction is achieved by a few support structures and routines having in mind an efficient platform for data loading, data forwarding, and data access by all nodes.

The Data Cyclotron delineation of its components leads to an architecture which can be integrated readily within an existing DBMS by injecting simple calls into the query execution plan. The performance penalty comes from waiting for parts of the hot-set to become available for local processing and the capability of the system to adjust quickly to changes in the workload set.

To achieve high throughput and low latency in the Data Cyclotron, the most relevant data for the current workload is identified and set with the highest priority to consume the available network bandwidth and storage ring space. Designated as hot-set, its decentralized management is responsible to assure fast access to the most relevant data and avoid data starvation for computations interested on data chunks with low relevance.

A ranking system is used to determine where a data chunk should be, i.e., in disk, memory, or storage ring. The ranking system is composed of ranking metric called the level of interest (*LOI*), and a threshold called the level of interest threshold (*LOIT*). *LOI* describes how popular was, is a data chunk for the current workload. A data chunk with low rank is more likely to be evicted then a data chunk with high rank.

Using a well known network simulator (*NS-2*), different approaches and methods were explored to reach the most efficient and robust solution to rank the data chunks popularity and define an optimal hot-set. With different scenarios, the results confirm our intuition that a storage ring based on the hot-set can achieve high throughput and low latency.

The experiments showed that a decentralized hot-set management can lead to an unfair management of less popular data when the storage ring is overloaded. To circumvent the problem a new state for the data was created, the *warm-set*. It contributes for a more precise definition of hot-set which is now refined to: a set composed of data chunks with the highest probability of utilization and optimal size for efficient network bandwidth usage, the highest throughput, and the lowest data access latency. Furthermore, an innovative cache management was designed to reduce latency for applications where instructions have dependencies and cannot be eligible for execution based only on input data from the storage ring.

The architecture (cf.,Chapter 3) together with the hot-set management (cf., Chapter 4) were conceptualized into a full functional system, DaCyDB (cf., Section 5.2). DaCyDB is the integration of a MonetDB instance on each Data Cyclotron node. MonetDB was chosen for being a column-store which are known for being efficient for intense data analysis, but also for using the operator-at-the-time paradigm and partitioned execution for efficient parallelism. DaCyDB is in favor of partitioned execution since it offers much better opportunities for speedup and scaleup then *pipeline* execution. By taking the large relational operators and partitioning their inputs and outputs, it is possible to use divide-and-conquer to turn one big job into many independent little ones and distribute them among nodes for processing. This is an ideal situation for speedup and scaleup [46].

The extended plans created for partitioned execution are used by DaCyDB to maximize the cooperative actions on the shared data chunks. Such cooperative work is only possible due to the strategy taken by MonetDB to unroll the loops through code repetition (cf., Section 5.2.3) which combined with code re-utilization, such as recycling [84, 83], turns the overhead of defining and executing long plans negligible. This cooperative access to the partitions together with the internal query parallelism contributes for high throughput and low query response time.

The DaCyDB validation and evaluation (cf., Section 5.3) was done in two different clusters for real workloads and different scenarios. The challenges and issues to cope with different bandwidths, traffic jams, un-balanced data distribution, and a flexible query parallelism to exploit a continuous data stream were identified and solutions proposed. Solutions to make DaCyDB more efficient for complex computations such as TPC-H queries. From a better plan optimization to an efficient management of intermediates to reduce virtual memory utilization, several improvements were considered and some tested with experiments. Furthermore, we delineated a new scheduler for an effortless and efficient intra and inter query parallelism.

At the same time, the protocols and routines presented for the hot-set management (cf., Chapter 4) were also validated. The evaluation was not intended to benchmark our novel architecture against contemporary approaches of single/parallel solutions such as MapReduce solutions. The engineering optimization space for the DaCyDB is largely unexplored and experience gained to date with the architecture would not do justice to such a comparison. Furthermore, compared to MapReduce solutions, DaCyDB design is intended for a different data processing stage (cf., Section 6.9).

From the three data processing stages: data collection, data preparation, and data presentation [58], MapReduce frameworks were designed to be efficient on the data preparation stage while DaCyDB was designed for the data presentation stage (cf., Section 6.9.2). The MapReduce paradigm was designed for slow networks and exploit a distributed file system for fault tolerance. It provides data parallelism instead of process parallelism. It is tailored to scale out in a cluster of commodity machines. On the other hand, the DaCyDB uses fast network to share and access storage nodes. Its design is based on the new trends for data centers while system like Hadoop were designed for traditional data center composed of commodity machines. Nevertheless, both solutions can also be adapted to somehow be efficient on others data processing stages.

The DaCyDB efficient resource utilization and utilization of scale up and scale out as two complementary solutions turns DaCyDB into an efficient architecture for complex computations on hybrid clusters. Giving priority to in-memory processing and overlapping communication with data processing (a requirement to save energy [36]), it joins the *green movement* for data centers. At the same time, it aligns with the hardware trends for data centers such as diskless nodes.

The idea of data movement between network nodes is the key for all this flexibility and efficiency. The absence of static data allocation is exploited to explore different algorithms for distributed processing. Computations are split into sub-computations. They are sent to the ring and each of them settles on a different node following the basic procedures of a normal computation. They are processed concurrently and the individual intermediate results are then combined to form the final computation result. Such freedom gives the grounds for an application to scale out without be bounded to any complex scheduling algorithm or data distribution scheme.

Furthermore, a computation is not tied to be executed to any specific node or group of nodes. Instead, each computation searches a lightly loaded node to execute on; the data needed will pass by upon request. This way, the load is not spread based on data assignment, but purely on the node's characteristics and on the storage ring load characteristics. This innovative and simple strategy intends to avoid hot spots that result from errors in the data allocation and query plan algorithms.

Based on the levels of utilization, each node decides to leave the ring or request a ring extension. This autonomous and dynamic adaption to accommodate the workload requirements is what defines *pulsating rings*. The pulsating rings provide an outlook for query processing workloads that can not be addressed with database sharding and map-reduce. It is a new concept to design fully flexible distributed query processing architectures. Moreover, it is conceivable that multiple pulsating rings live in the same physical cluster. At any point in time, such a mesh consists of numerous overlapped pulsating rings. The model exploits the independence and autonomy of each individual where everyone, based on the workload flow, works for the same goal, high throughput.

7.2 Research directions

The Data Cyclotron opens a vista on a research landscape of novel ways to implement distributed query processing. Cross fertilization from distributed systems, hardware trends, and analytic modeling in ring-structured services seems prudent. Likewise, the query execution strategies, the algorithms underpinning the relational operators, the query optimization strategies and updates all require a thorough re-evaluation in this context.

In this Section we lay down the most promising research directions for the DaCyDB and the Data Cyclotron architecture. Starting with DaCyDB, an interesting direction is the integration of updates using a version scheme to support different levels of result accuracy, and thus allow a database to be queried while it is updated. To understand the potentials of DaCyDB architecture, we plan a performance evaluation on super-computers to explore an aggressive solution to exploit iterative loading and diskless nodes as well as verify CPU affinity for efficient query parallelism. As last direction, the Chapter explains how easy is the integration of a generic row-store with the Data Cyclotron.

7.2.1 The space for updates.

An update can be classified into two categories, appends or records updates. For the first category, the append happens at the source nodes. One or more partitions are added to the source nodes and an update catalog request is sent around the ring by the node executing the update query. Once the request returns, the columns/table new version becomes official. The data is only loaded to the ring upon request using iterative loading (cf., Section 6.2.4). It is the most common situation for the application scenarios for which DaCyDB is targeted, i.e., the updates on data sets from eScience or data warehouses are mainly appends.

For a record update, the update is split into two phases, selection phase and update phase. The selection phase identifies the data chunks which contain the records to be updated. The selection statement is extracted from the update query to identify the data chunks. The data chunks are then requested to perform the update. A low granularity update to avoid the request of all partitions of a column.

An append of few records to each column is seen as a special record update. It avoids the creation of many partitions at the source nodes. In this case, the last partition is always the requested one. In case it is full, a new partition is created. For both cases, the update or a copy of the partition is sent to the source nodes to propagate the changes to the backups.

Distribute updates.

All update queries enter the ring at the same node. They are tagged with a timestamp to create a time order for all update messages. Their execution is however distributed. Hence, to have order preservation and data consistency DaCyDB uses database versioning. The versions control is done through a distributed lock. For each table a lock is put in circulation in the counter clockwise ring at the table's creation time.

The update queries are picked first-in-first-out order by the nodes. The node who picks an update query for execution uses the timestamp to add a new version to the table's lock. For each version a table's lock is at one of three phases: *selection* phase (the data chunks were requested), *update* phase (the node is updating the data chunks), or *done* phase (the update succeeded and a new version was established). To speed up the first phase, an update request has more priority than a read request, therefore, a data chunk is loaded into the hot-set at the reception of an update request, i.e., it does not need more requests to be popular enough to enter the hot-set.

In the first phase, for each version, the data chunks IDs are added to the table's lock. Like this, for disjoint updates over the same table, the *update* phase is done in parallel. However, simultaneous *done* phases per data chunks are not allowed. When the latter state is set, the node which has the updates for the next version on the same data chunks can proceed.

The node who updated the data chunks with version V is responsible to load the new version into the storage ring upon request. The data chunks are only allowed to be loaded once the lock with status *done* for version V has completed a full cycle, this is, all nodes are aware of version V and old copies from other nodes cache were invalidated. It is equivalent to a distributed commit of an update transaction.

Older versions are kept around and their metadata is stored in the tables locks and at the catalog of each node. Their maintenance is similar to the rest of the database, i.e., using its *LOI* each data chunk version bounces between the cold-set and hot-set. Once it expires it is dropped from the database.

In an ideal situation, a new version is defined and a *logical copy* is created without data duplication. In DaCyDB exists partial duplication. In case a data chunk Chk is modified, a new materialized view Chk' with the updates is created. Hence, there is some data redundancy between the Chk and Chk'. If not modified, the data chunk is used for several consecutive versions. It reduces the amount of resources to store several database versions. Furthermore, it reduces the number of data chunks flowing in the storage ring.

Results accuracy.

A database version represents a unique, consistent view of the database, distinguished from other versions by time and an unique identifier. In the context of DaCyDB, the user does not specify a specific version, it uses a time reference, or an interval, to define on which version should the query be executed. It is a light versioning distributed database compared to Hbase [73] or BigTable from Google [33].

With distributed updates, depending on the number of records to be updated, several read-only queries can start between the update announcement, i.e., its registration in the lock, and its commit. DaCyDB explores the updates timestamp to allow a user to define the result accuracy of his queries. The concept is used in *bitemporal* databases such as TimeDB [131].

A query at registration time is tagged with a timestamp and through a time reference indicates the accuracy of its result. For example, for query Q the result is considered accurate if it uses the last stable version at its registration time. Based on this information, DaCyDB inspects the locks flowing around the ring to determine the table's version Vt. Vt is then used on the request calls to collect the correct data chunks version from the storage ring.

An optimistic update approach is also possible. Since the data chunks for each version are not allowed to be loaded until it reaches the *done* stage, DaCyDB can pick the latest version registered five minutes before Q registration V' and send the query for execution. In mean time, if the update for V' is aborted, the query is restarted and a previous version is selected and the same steps are followed until a stable version is founded. Such approach allows the user to query a database while it is being updated.

A similar solution for centralized systems at the records level was presented in [111]. The work contains the *recipes* to build a transient versioning system. It contains methods for efficient and flexible transient versioning of records to avoid locking by read only queries. Like [20] it is a finer grain solution, i.e., they work at the record level. The same for the *Broadcast Disks multiversioning* model to handle updates [124], this is, it works at page level. The versioning model for DaCyDB works at data chunk level. It is intended for large data sets and distributed data processing. Nevertheless, the concepts and ideas described in [111, 124] should re-evaluated and re-considered for the DaCyDB context.

7.2.2 Supercomputers

The two clusters used for evaluation of DaCyDB (cf., Section 5.3.1) follow the standard design of modern clusters for distributed data analyses. They represent the near future of a cluster composed by commodity machines. The nodes characteristics have imposed some challenges and requested some conservative solutions to achieve low query response time and high throughput. The lack of memory for intermediates and the communication speed among the nodes required the definition of an efficient hot-set management to make sure the most relevant data is in circulation. What would happen if the resources available per machine are twenty time bigger, i.e., the dream cluster?

The dream cluster for DaCyDB would be a cluster composed by nodes with huge non-volatile main memories, such as STT-MRAM (spin-torque-transfer magnetoresistive random-access memory), network links reaching dozens of GB /sec to interconnect them and access independent storage nodes with an efficient iterative loading service as described in Section 6.2.4. Such luxury of resources would emphasize even more the DaCyDB advantages for distribute query processing.

To study the system behavior in such luxury conditions, to learn how to explore them and to design an aggressive architecture, we intend to evaluate DaCyDB in the Huygens cluster [1] from our national computer center [2]. Each node in Huygens has 16 dual cores, resulting in 32 cores per node. 92 of them has 128 GB of memory (4 GB/core) and 16 nodes with 256 GB memory (8 GB/core). Each node has 2 HCA (host channel adapter) and each HCA has 4x DDR InfiniBand ports, this is, 160 Gb/sec (20 GB/sec) inter-node bandwidth.

Pure in-memory data processing.

Using a ring of 16 nodes, the ones with 256 GB main memory, we intend to study how DaCyDB would behave with diskless nodes, i.e., no cold-set. In this situation, there is only data resident in memory. For the hot-set management, the data instead of being unload to the cold set, it would simply be dropped. Once the data becomes relevant enough to be loaded to the hot-set, the data chunks would be requested from the source nodes exploiting the large network bandwidth and the iterative loading service.

The data chunks would be retrieved in parallel from the storage nodes (defined using the machines from the 92 nodes set), and their load to the ring would be spread among a sub-set of nodes to avoid tension on the buffers of a single node. The system would be split into two independent layers, the storage layer and the computational

[1] https://www.sara.nl/systems/huygens

[2] https://www.sara.nl

layer. A split to efficiently allocate specialized resources for I/O tasks or computational tasks.

The strategy is shared by distributed processing systems which retrieve data from the cloud such as Daytona [3]. Daytona is a kind of MapReduce framework from Microsoft for which the input data is stored in the cloud. It exploits a stream of data from Azure cloud service to feed the computational nodes which are running map and reduce tasks.

New parallelism techniques.

Another aspect to be explored in Huygens is the design or utilization of new techniques for parallel processing. Exploiting its SMP (Symmetric Multiprocessing) nodes we intend to explore different techniques, such as CPU affinity, to design new scheduling algorithms.

In a SMP node each core is split into two virtual CPUs, leading to 64 (virtual or logic) CPUs per node. With 256 GB of main memory, each virtual CPU gets 4 GB for input data and intermediates. To each of them, independent *DataFlow* blocks or sub-computations are scheduled for execution. A request from a sub-computation Sf makes the data chunk to be stored in the memory region assigned to the virtual CPU executing Sf.

For CPU affinity, the sub-functions are scheduled using the MapReduce paradigm for multi-core machines such as Phoenix library [125] or Metis [106]. These libraries use CPU cores as nodes and schedule the computations in the same matter as MapReduce. Through data requests DaCyDB does not need an optimized structure, such as a hash with B+tree, to efficiently shuffle between cores, this is, from the mappers to the reducers.

The approach would dramatically improve memory throughput as long as the data is localized to specific processors. On the downside, it is expensive to move data from one processor to another, as in workload balancing. Therefore, the solution is attractive for workloads where the same sub-function is used for different data sub-sets or computations composed by several sub-functions interested on different data chunks and few dependencies, a typical MapReduce computation.

Large rings composed by SMP nodes.

The 64 virtual CPUs could be seen as normal DaCy nodes. With 16 nodes from

[3] http://research.microsoft.com/en-us/projects/daytona/

Huygens [4], a ring could be composed by 1024 virtual nodes. Furthermore, with InfiniBand DDR on PCI Express version 2.0, network loopback has a performance near to a shared message passing implementation. Authors in [98] have shown that for InfiniBand QDR it outperforms shared memory message passing implementation.

The large ring would be organized in two rings, the inner-ring and outer-ring (cf., Section 6.5.3 and Figure 6.12).

" A inner-ring is simplified version of a chordal ring, a simple ring with cross or chordal links between nodes on opposite sides. With different dimensions a chordal ring is used to define several virtual rings to speed up data forwarding of a single hot-set in a large ring. "

In this case, the inner-ring is the physical connection between the 16 peers. The outer-ring is defined by the 1024 nodes. Using loop-back, scheduling data movements between the node within the same peer is trivial. Through the requests catalog a local node is identified and a data chunk can be placed directly in its memory space with a single hop. Furthermore, the first peer's node is used as the connector to the inner-ring. If a data chunk is not required by any of the peer's nodes, the data chunk is automatically forwarded to the next peer. It reduces data access latency by avoiding the hop of 63 nodes before it reaches the next peer.

With this soft virtualization model, DaCyDB is able to operate with 1024 simple virtual nodes, connected through an efficient Data Cyclotron layer, and achieve high performance. Furthermore, the study of this approach would open the ground for an integration of the Data Cyclotron architecture with a MapReduce framework for efficient data access and data distribution between map and reduce phases, or even independent MapReduce jobs such as in Pig.

Blue Gene.

Blue Gene is another type of computer which could raise interest since they are known for their computational power. Blue Gene is an IBM project aimed at designing supercomputers that can reach operating speeds in the PFLOPS (petaFLOPS) range, with low power consumption. The major characteristic of these computers is the interconnection of nodes, a five-dimensional torus interconnect the compute nodes in a peer-2-peer mode. However, the architecture of a Blue Gene, such as Blue Gene/Q, is not suited for the Data Cyclotron application scenarios.

A Blue Gene/Q cluster is composed of compute nodes and storage nodes. Each

[4]To distinguish them from the virtual nodes from now on they are referred as peers.

compute chip gets contains 10 links with 2 GB/sec each (they assure 4 GB/sec bi-directional bandwidth), plus one extra 2 GB/sec link to with I/O nodes. This means each compute node is able to receive and send 20 GB of data per second. The bandwidth is attractive, however, each node only has 16 GB of main memory. It is an architecture balanced for intense computational applications and with low memory footprint such as floating point algorithms.

7.2.3 Row-store integration

The Data Cyclotron integration has a modest impact on existing query execution engines and distributed applications. The integration is restricted to data type independent exchange and three calls to define which data is needed, when it is needed, and when it is released. The DaCyDB prototype showed how easy it was to integrate the Data Cyclotron software with a column-store. To consummate such a fact we here describe the integration with a typical row-store DBMS.

A generic row store DBMS is composed of many components, but it is at the back-end where iteration over the data takes place. In these systems, the usual query life-cycle involves the following stages: the parser, rewriter, planner, and executor. The parser creates a parse tree using these object definitions and passes it to the rewriter. The rewriter retrieves any rules specific to database objects accessed by the query, rewrites the parse tree using these rules, and passes the rewritten parse tree to the planner. The planner, or the optimizer, finds the most optimal path for the query execution by looking at collected statistics or some other metadata. An execution plan is then passed to the executor. The main function of the executor is to fetch data needed by the query and pass the query results to the client.

An execution plan is composed of several operators such as scan methods, join methods, aggregation operations, etc. Generally, the operators are arranged in a tree as illustrated in Figure 7.1. The execution starts at the root of the tree and moves down to the next level of operators in case the input is not available. Hence, the data flows from the leaves towards the root and the output of the root node is the query result.

The first rows for processing, i.e., the raw-data, are fetched at the bottom of the tree by the scan operators. A scan operator receives the identification of the table and starts reading data from a storage device. The rows are retrieved one at the time or in batches.

Integration.

The planner will be the one responsible to insert calls into the query plan based on the catalog built during the data distribution among the nodes. To keep the interaction with the Data Cyclotron transparent for all operators, the most reasonable place for the

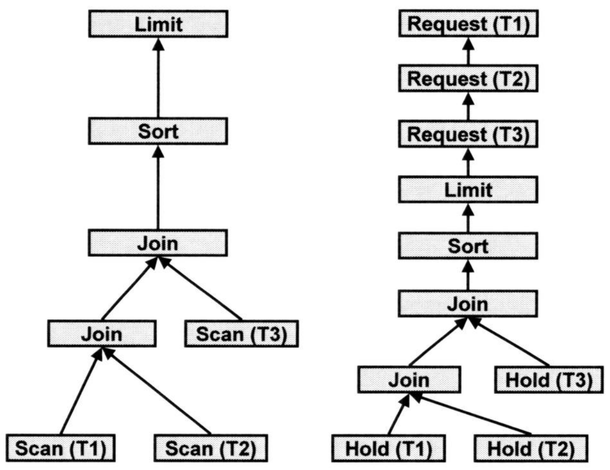


Figure 7.1: Typical execution plan in row-store DBMS.

"umbilical cord" between the DBMS and the Data Cyclotron is at the root and the bottom of the tree.

The requests will be injected at the root of a tree to inform the Data Cyclotron up front about all input tables. For long plans these injections are pushed down into the root of sub-trees to avoid data floods in the ring.

Due to the nature of the scan operator, the *pin()* and *unpin()* calls will be embedded into a single operator called *hold()* which would replace the scan operator in the plan. The *hold() pins* the table once the first row is requested and *unpins* once the last row is fetched.

The interaction between the row-store DBMS and the Data Cyclotron will have the cooperative work between the queries as described for MonetDB. The intra-query parallelism will also be explored using data partitioning.

Data partitioning.

Access to a full table is not feasible within a single buffer. Hence, a main table has to be sliced into sub-tables where each of them fits into a DaCy buffer. As for MonetDB, the tables will be sliced horizontally and then distributed uniformly among all nodes.

Depending on the application, horizontal partitioning might not be enough to define an optimal hot-set. For a workload with queries using a few columns from each table, the data-chunks would carry unused data, wasting bandwidth and memory bus.

Vertical partitioning, as a column-store, emerges as the right solution, however, to still explore the features and advantages of row-stores, the number of columns on each partition should then be defined based on the workload. Similar to RCFiles used in MapReduce, the tables would then be sliced horizontally based on value ranges, but each partition would then be vertically sliced into sub-groups of columns.

The ideal scenario would be a partition scheme, that, based on the workload, re-adapts the partitions over time. For example, creating vertical sub-partitions out of existing partitions, join sub-partitions into a single partition, or move columns from one partition to another. The discussion of this optimization is under research as well as the integration of a row-store, such as PostGres, with the Data Cyclotron.

7.3 Looking back, to look forward

The Data Cyclotron (DaCy) allied with novel and pioneer column store technology created DaCyDB. It addresses the grand challenge for distributed data processing: a self-organizing architecture which exploits all hardware resources for the current workload, achieves an accurate database subset definition, minimizes response time, and maximizes throughput without a single point for global co-ordination.

With an emphasis on simplicity and the autonomous behavior of each component, old concepts were revived, controversial methods were adopted, and orthogonal ideas were combined in novel ways. The presented, discussed, and fully functional prototyped architecture is the result of a thoroughly research exercise to define a new research direction for distributed data processing. It confirms our conviction that this type of challenging research is the right approach to reach new optimums in the database world.

Bibliography

[1] Daniel J. Abadi, Samuel R. Madden, and Nabil Hachem. Column-stores vs. row-stores: how different are they really? In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, pages 967–980. ACM, 2008.

[2] Azza Abouzeid, Kamil Bajda-Pawlikowski, Daniel Abadi, Avi Silberschatz, and Alexander Rasin. Hadoopdb: an architectural hybrid of mapreduce and dbms technologies for analytical workloads. *Proc. VLDB Endow.*, 2(1):922–933, August 2009.

[3] Azza Abouzied, Kamil Bajda-pawlikowski, Jiewen Huang, Daniel J Abadi, and Avi Silberschatz. Hadoopdb in action : Building real world applications. *Business*, page 11111114, 2010.

[4] Swarup Acharya, Rafael Alonso, Michael Franklin, and Stanley Zdonik. Broadcast Disks: Data Management for Asymmetric Communication Environments. In *SIGMOD '95*, pages 199–210, 1995.

[5] Swarup Acharya, Michael Franklin, and Stanley Zdonik. Balancing push and pull for data broadcast. In *SIGMOD '97*, pages 183–194, 1997.

[6] Foto N. Afrati and Jeffrey D. Ullman. Optimizing joins in a map-reduce environment. In *Proceedings of the 13th International Conference on Extending Database Technology*, EDBT '10, pages 99–110. ACM, 2010.

[7] Demet Aksoy, Michael J. Franklin, and Stanley B. Zdonik. Data staging for on-demand broadcast. In *VLDB '01*, pages 571–580, 2001.

[8] Samer Al-Kiswany, Dinesh Subhraveti, Prasenjit Sarkar, and Matei Ripeanu. Vmflock: virtual machine co-migration for the cloud. In *Proceedings of the 20th international symposium on High performance distributed computing*, HPDC '11, pages 159–170. ACM, 2011.

[9] Ioannis Alagiannis, Renata Borovica, Miguel Branco, Stratos Idreos, and Anastasia Ailamaki. Nodb: efficient query execution on raw data files. In *Proceedings of the 2012*

international conference on Management of Data, SIGMOD '12, pages 241–252. ACM, 2012.

[10] David G. Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. Fawn: a fast array of wimpy nodes. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 1–14. ACM, 2009.

[11] Subi Arumugam, Alin Dobra, Christopher M. Jermaine, Niketan Pansare, and Luis Perez. The datapath system: a data-centric analytic processing engine for large data warehouses. In *SIGMOD '10*, pages 519–530, New York, NY, USA, 2010. ACM.

[12] Shivnath Babu. Towards automatic optimization of mapreduce programs. In *Proceedings of the 1st ACM symposium on Cloud computing*, SoCC '10, pages 137–142, 2010.

[13] Pavan Balaji. Sockets vs rdma interface over 10-gigabit networks: An in-depth analysis of the memory traffic bottleneck. In *In RAIT workshop 04*, 2004.

[14] Mario Baldi and Yoram Ofek. A comparison of ring and tree embedding for real-time group multicast. *IEEE/ACM Trans. Netw.*, 11(3):451–464, 2003.

[15] Sujata Banerjee and Victor O. K. Li. Evaluating the distributed datacycle scheme for a high performance distributed system. *Journal of Computing and Information*, 1, 1994.

[16] Kenneth C. Barr and Krste Asanović. Energy-aware lossless data compression. *ACM Trans. Comput. Syst.*, 24(3):250–291, 2006.

[17] Luiz André Barroso and Urs Hölzle. The case for energy-proportional computing. *Computer*, 40(12):33–37, 2007.

[18] Iwona Bialynicka-Birula. Hadoop dont's: What not to do to harvest hadoop's full potential. `http://www.databonanza.com/2011/05/hadoop-donts-what-not-to-do-to-harve%st.html`, 2011.

[19] Spyros Blanas, Jignesh M. Patel, Vuk Ercegovac, Jun Rao, Eugene J. Shekita, and Yuanyuan Tian. A comparison of join algorithms for log processing in mapreduce. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, SIGMOD '10, pages 975–986. ACM, 2010.

[20] Paul M. Bober and Michael J. Carey. On mixing queries and transactions via multiversion locking. In *Proceedings of the Eighth International Conference on Data Engineering*, pages 535–545. IEEE Computer Society, 1992.

[21] P. Boncz, M. Kersten, and S. Manegold. Breaking the Memory Wall in MonetDB. *Commun. ACM*, 51(12):77–85, 2008.

[22] Peter Boncz, Stefan Manegold, and Martin Kersten. Optimizing Main-Memory Join On Modern Hardware. *IEEE Transactions on Knowledge and Data Engineering*, 14(4):709 – 730, July 2002.

[23] R. H. Bordini, J. F. Hbner, and R. Vieira. Jason and the golden fleece of agent-oriented programming, 2005.

[24] Vinayak R. Borkar, Michael J. Carey, Raman Grover, Nicola Onose, and Rares Vernica. Hyracks: A flexible and extensible foundation for data-intensive computing. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering*, ICDE '11, pages 1151–1162, 2011.

[25] Sumit Kumar Bose, Srikumar Krishnamoorthy, and Nilesh Ranade. Allocating resources to parallel query plans in data grids. In *GCC '07*, pages 210–220, 2007.

[26] S. Bourduas and Z. Zilic. A hybrid ring/mesh interconnect for network-on-chip using hierarchical rings for global routing. In *Proceedings of the First International Symposium on Networks-on-Chip*, NOCS '07, pages 195–204. IEEE Computer Society, 2007.

[27] T. F. Bowen, G. Gopal, G. Herman, T. Hickey, K. C. Lee, W. H. Mansfield, J. Raitz, and A. Weinrib. The datacycle architecture. *Commun. ACM*, 35(12):71–81, 1992.

[28] Sebastian Bre, Eike Schallehn, and Ingolf Geist. Towards optimization of hybrid cpu/gpu query plans in database systems. In *New Trends in Databases and Information Systems*, volume 185 of *Advances in Intelligent Systems and Computing*, pages 27–35. Springer Berlin Heidelberg, 2013.

[29] Lionel Brunie and Harald Kosch. Modelization and simulation of parallel relational query execution plans using dpl graphs and high-level petri nets, 1996.

[30] Robin Rehrmann Carsten Binnig, Franz Faerber and Rudolf Riewe. Funsql: It is time to make sql functional. In *Proceedings of the 15th International Conference on Extending Database Technology*, EDBT '12. ACM, 2012.

[31] Cassandra. Cassandra. `http://cassandra.apache.org/`.

[32] Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert R. Henry, Robert Bradshaw, and Nathan Weizenbaum. Flumejava: easy, efficient data-parallel pipelines. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '10, pages 363–375. ACM, 2010.

[33] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: a distributed storage system for structured data. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*, pages 205–218. USENIX Association, 2006.

[34] Surajit Chaudhuri and Vivek R. Narasayya. Self-tuning database systems: A decade of progress. In *VLDB '07*, pages 3–14, 2007.

[35] Xuedong Chen, Patrick E. O'Neil, and Elizabeth J. O'Neil. Adjoined dimension column clustering to improve data warehouse query performance. In *ICDE*, pages 1409–1411, 2008.

[36] Yanpei Chen, Sara Alspaugh, Dhruba Borthakur, and Randy Katz. Energy efficiency for large-scale mapreduce workloads with significant interactive analysis. In *Proceedings of the 7th ACM european conference on Computer Systems*, EuroSys '12, pages 43–56. ACM, 2012.

[37] Byung-Gon Chun, Gianluca Iannaccone, Giuseppe Iannaccone, Randy Katz, Gunho Lee, and Luca Niccolini. An energy case for hybrid datacenters. *SIGOPS Oper. Syst. Rev.*, 44(1):76–80, 2010.

[38] Tam Chun. Performance studies of high-speed communication on commodity cluster. Thesis, 2001.

[39] Deep Cloud. Deep cloud. `http://deepcloud.co/web1/`.

[40] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, Khaled Elmeleegy, and Russell Sears. Mapreduce online. In *Proceedings of the 7th USENIX conference on Networked systems design and implementation*, pages 21–21. USENIX Association, 2010.

[41] Intel Corporation. Intel previews intel xeon nehalem-ex processor. `http://www.intel.com/pressroom/archive/releases/2009/20090526comp.htm`, 2009.

[42] Intel Corporation. Intel details 2011 processor features. `http://newsroom.intel.com/community/intel_newsroom/blog/2010/09/13/inte%l-details-2011-processor-features-offers-stunning-visuals-built-in`, 2011.

[43] CouchDB. Couchdb. `http://couchdb.apache.org/`.

[44] Sudipto Das, Shoji Nishimura, Divyakant Agrawal, and Amr El Abbadi. Albatross: lightweight elasticity in shared storage databases for the cloud using live data migration. *Proc. VLDB Endow.*, pages 494–505, 2011.

[45] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[46] D. DeWitt and J. Gray. Parallel Database Systems: The Future of High Performance Database Systems. *Communications of the ACM*, 35(6):85–98, 1992.

[47] Jens Dittrich, Jorge-Arnulfo Quiané-Ruiz, Alekh Jindal, Yagiz Kargin, Vinay Setty, and Jörg Schad. Hadoop++: making a yellow elephant run like a cheetah (without it even noticing). *Proc. VLDB Endow.*, pages 515–529, 2010.

[48] EcoDB. Ecodb. `http://pages.cs.wisc.edu/~jignesh/ecodb/`.

[49] Marwa Elteir, Heshan Lin, and Wu-chun Feng. Enhancing mapreduce via asynchronous data processing. In *Proceedings of the 2010 IEEE 16th International Conference on Parallel and Distributed Systems*, ICPADS '10, pages 397–405. IEEE Computer Society, 2010.

[50] Xiaobo Fan, Wolf-Dietrich Weber, and Luiz Andre Barroso. Power provisioning for a warehouse-sized computer. In *Proceedings of the 34th annual international symposium on Computer architecture*, ISCA '07, pages 13–23. ACM, 2007.

[51] Avrilia Floratou, Jignesh M. Patel, Eugene J. Shekita, and Sandeep Tata. Column-oriented storage techniques for mapreduce. *Proc. VLDB Endow.*, pages 419–429, 2011.

[52] Michail D. Flouris and Evangelos P. Markatos. The network ramdisk: Using remote memory on heterogenous nows, 1998.

[53] A. Foong, T. Huff, H. Hum, J. Patwardhan, and G. Regnier. TCP performance re-visited. In *Proceedings of the 2003 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 70–79, 2003.

[54] P. Frey, R. Goncalves, M. Kersten, and J. Teubner. Spinning relations: High-speed networks for distributed join processing. In *DaMoN '09*, pages 27–33, 2009.

[55] P. Frey, R. Goncalves, M. Kersten, and J. Teubner. A spinning join that does not get dizzy, 2010.

[56] Philip Frey. *Zero-Copy Network Communication: An Applicability Study of iWARP beyond Micro Benchmarks*. PhD thesis, Department of Computer Science, ETH, Zurich, Zurich, Switzerland, 2010.

[57] Philip W. Frey and Gustavo Alonso. Minimizing the hidden cost of RDMA. In *ICDCS '09*, pages 553–560, 2009.

[58] Alan Gates. Pig and hive at yahoo! `http://developer.yahoo.com/blogs/hadoop/posts/2010/08/pig_and_hive_at_y%ahoo/`, 2010.

[59] Alan F Gates. Overview of hadoop. `http://ofps.oreilly.com/titles/9781449302641/hadoop_overview.html`.

[60] Alan F. Gates, Olga Natkovich, Shubham Chopra, Pradeep Kamath, Shravan M. Narayanamurthy, Christopher Olston, Benjamin Reed, Santhosh Srinivasan, and Utkarsh Srivastava. Building a high-level dataflow system on top of map-reduce: the pig experience. *Proc. VLDB Endow.*, pages 1414–1425, 2009.

[61] Johann George. qperf. `http://linux.die.net/man/1/qperf`.

[62] Georgios Giannikis, Gustavo Alonso, and Donald Kossmann. Shareddb: killing one thousand queries with one stone. *Proc. VLDB Endow.*, 5(6):526–537, 2012.

[63] Romulo Goncalves and Martin Kersten. The data cyclotron query processing scheme. In *EDBT '10*, 2010.

[64] Google. Jaql. `http://code.google.com/p/jaql/`.

[65] Naga Govindaraju, Jim Gray, Ritesh Kumar, and Dinesh Manocha. Gputerasort: High performance graphics co-processor sorting for large database management. In *SIGMOD '06*, pages 325–336, 2006.

[66] Goetz Graefe. Database servers tailored to improve energy efficiency. In *Proceedings of the 2008 EDBT workshop on Software engineering for tailor-made data management*, SETMDM '08, pages 24–28. ACM, 2008.

[67] Jim Gray, Alex S. Szalay, et al. Data Mining the SDSS SkyServer Database. *MSR-TR-2002-01*, January 2002.

[68] H3C. Challenges and new trends in data center infrastructure. *White paper*, June 2009.

[69] Ismail Omar Hababeh, Muthu Ramachandran, and Nicholas Bowring. A high-performance computing method for data allocation in distributed database systems. *The Journal of Supercomputing*, pages 3–18, 2007.

[70] Hadoop. Distributed cache. `http://hadoop.apache.org/common/docs/r0.20.2/mapred_tutorial.html\#Dist%ributedCache`.

[71] Hadoop. hadoop. `http://hadoop.apache.org/`.

[72] Hadoop. Hadoop tutorial. `http://hadoop.apache.org/mapreduce/docs/r0.22.0/mapred_tutorial.html`.

[73] Hbase. Hbase. `http://hbase.apache.org/`.

[74] Abdelsalam A. Helal, Abdelsalam A. Heddaya, and Bharat B. Bhargava. *Replication Techniques in Distributed Systems*. KAP (Kluwer Academic Publishers), 1996.

[75] Gary Herman, K. C. Lee, and Abel Weinrib. The datacycle architecture for very high throughput database systems. *SIGMOD Rec.*, 16(3):97–103, 1987.

[76] Herodotos Herodotou, Harold Lim, Gang Luo, Nedyalko Borisov, Liang Dong, Fatma Bilgen Cetin, and Shivnath Babu. Starfish: A self-tuning system for big data analytics. In *CIDR 2011, Fifth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 9-12, 2011, Online Proceedings*, pages 261–272. www.crdrdb.org, 2011.

[77] Anthony J. G. Hey, Stewart Tansley, and Kristin M. Tolle. Microsoft Research, 2009.

[78] Ricky Ho. What hadoop is good at. `http://horicky.blogspot.nl/2009/11/what-hadoop-is-good-at.html`, 2009.

[79] Wei Hong and Michael Stonebraker. Optimization of parallel query execution plans in xprs. In *Proceedings of the first international conference on Parallel and distributed information systems*, PDIS '91, pages 218–225, Los Alamitos, CA, USA, 1991. IEEE Computer Society Press.

[80] Sotiris Ioannidis, Evangelos P. Markatos, and Julia Sevaslidou. On using network memory to improve the performance of transaction-based systems, 1998.

[81] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 59–72, 2007.

[82] M. Ivanova, N. Nes, R. Gonçalves, and M. L. Kersten. MonetDB/SQL Meets SkyServer: the Challenges of a Scientific Database. In *Proc. SSDBM*, Banff, Canada, July 2007.

[83] Milena Ivanova, Martin Kersten, Niels Nes, and Romulo Goncalves. An Architecture For Recycling Intermediates In A Column-Store. *ACM Transactions on Database Systems*, 35(4), 2010.

[84] Milena Ivanova, Martin Kersten, Niels Nes, and Romulo Gonçalves. An architecture for recycling intermediates in a column-store. In *SIGMOD '09: Proceedings of the 35th SIGMOD international conference on Management of data*, pages 309–320, New York, NY, USA, 2009. ACM.

[85] Eaman Jahani, Michael J. Cafarella, and Christopher Ré. Automatic optimization for mapreduce programs. *Proc. VLDB Endow.*, pages 385–396, 2011.

[86] Vijay Janapa Reddi, Benjamin C. Lee, Trishul Chilimbi, and Kushagra Vaid. Web search using mobile cores: quantifying and mitigating the price of efficiency. In *Proceedings of the 37th annual international symposium on Computer architecture*, ISCA '10, pages 314–325. ACM, 2010.

[87] Dawei Jiang, Beng Chin Ooi, Lei Shi, and Sai Wu. The performance of mapreduce: an in-depth study. *Proc. VLDB Endow.*, pages 472–483, 2010.

[88] Frans Kaashoek, Robert Morris, and Yandong Mao. Optimizing mapreduce for multicore architectures. *MIT, White Paper*, 2010.

[89] Tim Kaldewey, Eugene J. Shekita, and Sandeep Tata. Clydesdale: structured data processing on mapreduce. In *Proceedings of the 15th International Conference on Extending Database Technology*, EDBT '12, pages 15–25. ACM, 2012.

[90] Yagiz Kargin, Holger Pirk, Milena Ivanova, Stefan Manegold, and Martin Kersten. Instant-on scientific data warehouses —lazy etl for data-intensive research. In *Proceedings of the VLDB 2012 Workshop on Business Intelligence for the Real Time Enterprise*. ACM, 2012.

[91] M. Kersten, Y. Zhang, M. Ivanova, and N. Nes. Sciql, a query language for science applications. In *Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases*, pages 1–12. ACM, 2011.

[92] Martin L. Kersten. The Database Architecture Jigsaw Puzzle. In *ICDE '08*, pages 3–4, 2008.

[93] Martin L. Kersten, Stratos Idreos, Stefan Manegold, and Erietta Liarou. The researcher's guide to the data deluge: Querying a scientific database in just a few seconds. *PVLDB*, 4(12):1474–1477, 2011.

[94] John Kim and Hanjoon Kim. Router microarchitecture and scalability of ring topology in on-chip networks. In *Proceedings of the 2nd International Workshop on Network on Chip Architectures*, pages 5–10, New York, NY, USA, 2009. ACM.

[95] John Kim and Hanjoon Kim. Router microarchitecture and scalability of ring topology in on-chip networks. In *Proceedings of the 2nd International Workshop on Network on Chip Architectures*, NoCArc '09, pages 5–10. ACM, 2009.

[96] Shinya Kitajima, Tsutomu Terada, Takahiro Hara, and Shojiro Nishio. Query processing methods considering the deadline of queries for database broadcasting systems. *Syst. Comput. Japan*, 38(2):21–31, 2007.

[97] Kenneth C. Knowlton. A fast storage allocator. *Commun. ACM*, 8(10):623–624, 1965.

[98] Matthew J. Koop, Wei Huang, Karthik Gopalakrishnan, and Dhabaleswar K. Panda. Performance analysis and evaluation of pcie 2.0 and quad-data rate infiniband. In *Proceedings of the 2008 16th IEEE Symposium on High Performance Interconnects*, HOTI '08, pages 85–92. IEEE Computer Society, 2008.

[99] Donald Kossmann. The state of the art in distributed query processing. *ACM Comput. Surv.*, 32(4):422–469, 2000.

[100] Willis Lang and Jignesh M. Patel. Towards eco-friendly database management systems. In *CIDR*, 2009.

[101] Willis Lang, Jignesh M. Patel, and Srinath Shankar. Wimpy node clusters: what about non-wimpy workloads? In *Proceedings of the Sixth International Workshop on Data Management on New Hardware*, DaMoN '10, pages 47–55. ACM, 2010.

[102] Guillaume Leduc. Road traffic data: Collection methods and applications. *Technical Note: JRC 47967*, November 2008.

[103] Mong Li Lee, Masaru Kitsuregawa, Beng Chin Ooi, Kian-Lee Tan, and Anirban Mondal. Towards self-tuning data placement in parallel database systems. *SIGMOD Rec.*, 29(2):225–236, 2000.

[104] Jiuxing Liu, Dan Poff, and Bulent Abali. Evaluating high performance communication: a power perspective. In *Proceedings of the 23rd international conference on Supercomputing*, ICS '09, pages 326–337. ACM, 2009.

[105] Stefan Manegold, Martin Kersten, and Peter Boncz. Database Architecture Evolution: Mammals Flourished Long Before Dinosaurs Became Extinct. In *Proceedings of the International Conference on Very Large Data Bases (VLDB, 2009)*. VLDB, August 2009. 10-year Best Paper Award for Database Architecture Optimized for the New Bottleneck: Memory Access. In Proceedings of the International Conference on Very Large Data Bases (VLDB), pp 54-65, Edinburgh, United Kingdom, September 1999.

[106] Yandong Mao, Robert Morris, and M. Frans Kaashoek. Optimizing mapreduce for multicore architectures. *Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Tech. Rep*, 2010.

[107] Holger Märtens, Erhard Rahm, and Thomas Stöhr. Dynamic query scheduling in parallel data warehouses. *Concurrency and Computation: Practice and Experience*, 15(11-12):1169–1190, 2003.

[108] Joe McKendrick. Data center trends - the 12 top ways data management will evolve in the 2010s. `http://www.dbta.com`, 2010.

[109] Mellanox. Uda solution for hadoop. `http://www.mellanox.com/content/pages.php?pg=hadoop`.

[110] MemBase. Membase. `http://www.couchbase.com/membase`.

[111] C. Mohan, Hamid Pirahesh, and Raymond Lorie. Efficient and flexible methods for transient versioning of records to avoid locking by read-only transactions. In *Proceedings of the 1992 ACM SIGMOD international conference on Management of data*, SIGMOD '92, pages 124–133. ACM, 1992.

[112] Curt Monash. Hadapt (commercialized hadoopdb). http://www.dbms2.com/2011/03/23/hadapt-commercialized-hadoopdb/, 2011.

[113] MongoDB. Mongodb. http://www.mongodb.org/.

[114] Derek G. Murray, Malte Schwarzkopf, Christopher Smowton, Steven Smith, Anil Madhavapeddy, and Steven Hand. Ciel: a universal execution engine for distributed data-flow computing. In *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, NSDI'11, pages 9–9, 2011.

[115] null Shuang Liang, R. Noronha, and D.K. Panda. Swapping to remote memory over infiniband: An approach using a high performance network block device. *Cluster Computing, IEEE International Conference on*, pages 1–10, 2005.

[116] Tomasz Nykiel, Michalis Potamias, Chaitanya Mishra, George Kollios, and Nick Koudas. Mrshare: sharing across multiple queries in mapreduce. *Proc. VLDB Endow.*, 3(1-2):494–505, September 2010.

[117] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: a not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, pages 1099–1110. ACM, 2008.

[118] Pat O'Neil, Betty O'Neil, and Xuedong Chen. Start schema benchmark (ssb). www.cs.umb.edu/~poneil/StarSchemaB.PDF.

[119] Oracle. External tables concepts. http://docs.oracle.com/cd/B19306_01/server.102/b14215/et_concepts.htm.

[120] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazires, Subhasish Mitra, Aravind Narayanan, et al. *The Case for RAMClouds: Scalable High-Performance Storage Entirely in DRAM*. Stanford University, 2010.

[121] Apoorva Patel. The future of computation. *Workshop on Quantum Information, Computation and Communication*, March 2005.

[122] D. C. Pham, T. Aipperspach, D. Boerstler, M. Bolliger, R. Chaudhry, D. Cox, P. Harvey, P. M. Harvey, H. P. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Pham, J. Pille, S. Posluszny, M. Riley, D. L. Stasiak, M. Suzuoki, O. Takahashi, J. Warnock, S. Weitzel, D. Wendel, and K. Yazawa. Overview of the architecture, circuit design, and physical implementation of a first-generation cell processor. *Solid-State Circuits, IEEE Journal of*, 41(1):179–196, 2006.

[123] Pig. Pig. http://pig.apache.org/.

[124] Evaggelia Pitoura and Panos K. Chrysanthis. Exploiting versions for handling updates in broadcast disks. In *Proceedings of the 25th International Conference on Very Large Data Bases*, VLDB '99, pages 114–125. Morgan Kaufmann Publishers Inc., 1999.

[125] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, HPCA '07, pages 13–24. IEEE Computer Society, 2007.

[126] S. Ratnasamy et al. A Scalable Content-addressable Network. In *SIGCOMM '01*, pages 161–172, 2001.

[127] Suzanne Rivoire, Mehul A. Shah, Parthasarathy Ranganathan, and Christos Kozyrakis. Joulesort: a balanced energy-efficiency benchmark. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, SIGMOD '07, pages 365–376. ACM, 2007.

[128] Scilens. Platform. `http://www.scilens.org/content/platform`, 2012.

[129] Scott Sellers. Overcoming the big performance challenge of big data in finance. `http://blogs.computerworld.com/20084/overcoming_the_big_performance_cha%llenge_of_big_data_in_finance`, 2012.

[130] Nati Shalom. The tera-scale effect. `http://natishalom.typepad.com/nati_shaloms_blog/2010/11/the-tera-scale-%effect-part-i.html`, 2010.

[131] Andreas Steiner, Dipl Informatik ing Eth, M. C. Norrie, Prof Dr, Prof Dr, and C. A. Zehnder. A generalisation approach to temporal data models and their implementations, 1998.

[132] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM '01*, pages 149–160, 2001.

[133] Michael Stonebraker, Daniel Abadi, David J. DeWitt, Sam Madden, Erik Paulson, Andrew Pavlo, and Alexander Rasin. Mapreduce and parallel dbmss: friends or foes? *Commun. ACM*, 53(1):64–71, 2010.

[134] Michael Stonebraker, Paul M. Aoki, and Witold et al. Mariposa: a wide-area distributed database system. *The VLDB Journal '96*.

[135] Michael Stonebraker, Randy H. Katz, David Patterson, and John Ousterhout. The design of xprs. In *PROCEEDINGS OF THE 14TH INTERNATIONAL CONFERENCE ON VERY LARGE DATA BASES*, pages 318–330, 1988.

[136] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The end of an architectural era: (it's time for a complete rewrite). In *Proceedings of the 33rd international conference on Very large data bases*, VLDB '07, pages 1150–1160. VLDB Endowment, 2007.

[137] Alexander S. Szalay, Jim Gray, et al. The SDSS SkyServer: Public Access to the Sloan Digital Sky Server Data. In *SIGMOD*, pages 570–581, 2002.

[138] A. Thusoo, J.S. Sarma, N. Jain, Zheng Shao, P. Chakka, Ning Zhang, S. Antony, Hao Liu, and R. Murthy. Hive - a petabyte scale data warehouse using hadoop. In *Proceedings of the 2010 IEEE 26th International Conference on Data Engineering*, ICDE '10, pages 996–1005. IEEE Computer Society, 2010.

[139] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive: a warehousing solution over a map-reduce framework. *Proc. VLDB Endow.*, 2(2):1626–1629, August 2009.

[140] Niraj Tolia. Computer networking: What is a flattened butterfly network topology? `http://www.quora.com/Computer-Networking/What-is-a-Flattened-Butterfly-%network-topology`, 2010.

[141] Rares Vernica, Andrey Balmin, Kevin S. Beyer, and Vuk Ercegovac. Adaptive mapreduce using situation-aware mappers. In *Proceedings of the 15th International Conference on Extending Database Technology*, EDBT '12, pages 420–431. ACM, 2012.

[142] Xiao Wei Wang, Jie Zhang, Hua Ming Liao, and Li Zha. Dynamic split model of resource utilization in mapreduce. In *Proceedings of the second international workshop on Data intensive computing in the clouds*, DataCloud-SC '11, pages 21–30. ACM, 2011.

[143] Daniel Warneke and Odej Kao. Nephele: efficient parallel data processing in the cloud. In *Proceedings of the 2nd Workshop on Many-Task Computing on Grids and Supercomputers*, MTAGS '09, 2009.

[144] Wikipedia. Buddy memory allocation. `http://en.wikipedia.org/wiki/Buddy_memory_allocation`.

[145] Wikipedia. Chord peer-to-peer. `http://en.wikipedia.org/wiki/Chord_(peer-to-peer)`.

[146] Wikipedia. Database — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/wiki/Database`, 2004. [Online; accessed March-2012].

[147] Wikipedia. Space-based architecture. `http://en.wikipedia.org/wiki/Space-based_architecture`, 2004. [Online; accessed March-2012].

[148] Yahoo. Module 2: The hadoop distributed file system. `http://developer.yahoo.com/hadoop/tutorial/module2.html`.

[149] Hung-chih Yang, Ali Dasdan, Ruey-Lung Hsiao, and D. Stott Parker. Map-reduce-merge: simplified relational data processing on large clusters. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, SIGMOD '07, pages 1029–1040. ACM, 2007.

[150] C. T. Yu and C. C. Chang. Distributed Query Processing. *ACM Computing*, 16(4), 1984.

[151] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, HotCloud'10, pages 10–10, 2010.

[152] Daniel C. Zilio, Jun Rao, Sam Lightstone, Guy Lohman, Adam Storm, Christian Garcia-Arellano, and Scott Fadden. Db2 design advisor: Integrated automatic physical database design. In *VLDB '04*, pages 1087–1097, 2004.

[153] ZooKeeper. Zookeeper. `http://zookeeper.apache.org/`.

[154] M. Zukowski, P. A. Boncz, N. Nes, and S. Heman. MonetDB/X100 - A DBMS In The CPU Cache. *IEEE Data Engineering Bulletin*, 28:17–22, 2005.

[155] Marcin Zukowski, Sándor Héman, Niels Nes, and Peter Boncz. Cooperative scans: dynamic bandwidth sharing in a dbms. In *VLDB '07*, pages 723–734, 2007.

List of Figures

List of Tables

Summary

The grand challenge for distributed query processing is to come up with a self-organizing architecture which exploits all hardware resources for the current workload, defines an accurate database subset, minimizes response time, and maximizes throughput without a single point for global coordination.

The Data Cyclotron architecture addresses this grand challenge using turbulent data movement through a storage ring built from distributed main memory. It capitalizes the functionality offered by modern remote-DMA network hardware. Its design reconsiders the network cost model followed by most distributed systems, i.e., slow connections in the past.

Queries assigned to individual nodes interact with the Data Cyclotron by requesting and picking data fragments up from a database subset continuously flowing around, i.e., the *hot-set*. The *hot-set* is dynamically adjusted based on ring characteristics such as load and workload demands.

Composed by the most relevant data for the workload, the *hot-set* is kept as light as possible. Only fragments with a high cumulative query interest over the cycles are kept in the ring. Therefore, a change in the workload focus, i.e., new input data is required, the *hot-set* is re-adjusted. The readjustment is triggered by query requests for data access. The Data Cyclotron's self-organization in a distributed manner, keeping optimal resource utilization, gradually replaces the data that composes the *hot-set* to accommodate the current workload.

The Data Cyclotron leaves the decision to execute a query to the nodes, i.e., there is no global cost model for query assignment. It makes the system flexible to scale. It exploits the fact that each individual query can be processed at any node; we are sure that the relevant data will pass by. This way, the processing node does not need to be chosen with respect to data locality. Instead, performance factors crucial for the system at large are taken into account. Therefore, the Data Cyclotron, instead of moving the data to the queries (DDBMS) or a query to the data (peer-2-peer), it moves data and queries as two complementary entities. As far we are aware, it is an innovative and

simple strategy rarely, or maybe never, used before in the distributed query processing environments.

The approach is first illustrated using an extensive simulation study. The results underpin the *hot-set* management robustness in turbulent workload scenarios. Furthermore, a fully functional prototype of the proposed architecture has been integrated with a column-store. The integration was tested within a multi-rack cluster equipped with InfiniBand using both micro benchmarks and high-volume workloads based on the well-known decision support benchmark, TPC-H [5].

The results demonstrate the architecture feasibility and inspired the design of a novel solution for distributed query processing, the DaCyDB. The DaCyDB opens a new vista for modern distributed database architectures with a plethora of new research challenges. It uses the decentralized Data Cyclotron architecture to offer a seamless transition between a scale up solution to a scale out solution, or the best case scenario, have both in harmony. This seamless transition is a pillar to explore both models and achieve different levels of parallelism.

For efficient resource utilization and an absentee of central coordination, the DaCyDB exploits the Data Cyclotron *pulsating ring* instead of the master-slave model. A *pulsating ring* adaptivaly grows and shrinks to continuously seek the optimal number of nodes comprising a ring and guaranties the fastest data flow. All decisions are made locally at each node: which query to execute, when to leave, or when to split. Hence, it is possible to have a mesh of heterogeneous rings, and thus make the architecture flexible and robust to difference workload resource demands. The model exploits the independence and autonomy of each individual where everyone, based on the workload flow, works for the same goal, high throughput.

[5]http://www.tpc.org/tpch/

Samenvatting

Zou het niet fantastisch zijn als in een computernetwerk alle hardware naar volledige capaciteit gebruikt werd? Als je daardoor nog sneller een antwoord op je vraag kreeg? Als dit met zoveel mogelijk data tegelijkertijd ging? En als dit zou kunnen zonder dat het hele netwerk onder toezicht van één computer stond? Het is de droom van zelforganiserende computernetwerken, en de uitdaging voor gedistribueerde query-verwerking.

Deze droom wordt werkelijkheid met de Data Cyclotron-architectuur. De unieke wijze waarop gegevens, op maximale snelheid, worden rondgepompt haalt het beste uit de functionaliteit van moderne hardware. Door het gebruik van slechts computergeheugen en speciale zogeheten remote-DMA-netwerkkaarten, kunnen verouderde modellen voor netwerkkostenberekening op de schop.

Elke computer in de Data Cyclotron-architectuur kan een query afhandelen. Door de rondgepompte gegevens op het juiste moment op te pakken, kan elke computer om gegevens vragen en er vervolgens mee werken. De gegevens die steeds weer rondgaan vormen de zogenoemde hot-set. Door de grootte van de hot-set aan te passen aan de vraag die er in het hele netwerk naar de gegevens bestaat, kan op de meest efficiënte wijze met bijvoorbeeld het beschikbare geheugen worden omgegaan. Gegevens die zelden of niet nodig zijn worden zo niet langer onnodig rondgepompt. Maar ook andersom, als blijkt dat gegevens vaak nodig zijn, kunnen deze worden toegevoegd aan de hot-set. Dit zorgt ervoor dat de Data Cyclotron-architectuur zich aanpast aan het werk dat gedaan wordt in het netwerk.

Welke computer de taak van het uitvoeren van een query op zich neemt, is geheel aan de computers in het netwerk zelf. Er wordt geen model gebruikt dat vereist dat de computers in het netwerk met elkaar communiceren om uit te vinden wie een query het beste kan uitvoeren. Zo blijft de Data Cyclotron- architectuur flexibel en schaalbaar. Vanwege de unieke eigenschap dat gegevens almaar worden rondgepompt, kan elke computer elke query beantwoorden. Een query kan dus daar worden uitgevoerd waar dat voor het gehele netwerk de beste keuze is, ongeacht waar de gegevens zich in het netwerk bevinden. Het verplaatsen van zowel de gegevens als de queries is een nieuwe techniek. Het gaat verder dan traditionele verplaatsing van de gegevens naar de queries (gedistribueerde databases) of het verplaatsen van de query naar de gegevens (zogenaamde peer-2-peer netwerken), door beide technieken onafhankelijk van elkaar toe te passen.

De Data Cyclotron-architectuur is gerealiseerd op een bestaande column-store database, en onderworpen aan tests op micro- en macroniveau. Om het rondpompen van de gegevens goed te kunnen uitvoeren, is gebruikgemaakt van een speciaal computernetwerk uitgerust met hogesnelheids-InfiniBand-netwerkkaarten. Onder andere de de facto databasetest, TPC-H, is op deze manier succesvol getest met de implementatie van de Data Cyclotron-architectuur: DaCyDB.

De resultaten van DaCyDB geven aan dat de gekozen aanpak haalbaar is. Dit nieuw ontgonnen terrein biedt dan ook veel aanknopingspunten voor verder onderzoek.

Zo is bijvoorbeeld het vergroten of verkleinen van het aantal deelnemers die gegevens rondpompen een manier om de grootte van de database dynamisch te veranderen. Met het ontbreken van één centrale toezichthouder in het netwerk zijn alle computers vrij om zelf te bepalen in welke vorm ze aan zo'n ring deelnemen. Dit kunnen natuurlijk ook meerdere ringen per computer zijn, waardoor er een flexibel, maar bovenal zeer efficiënt systeem kan worden gecreëerd, dat het beste uit de hardware haalt.

Acknowledgments

The conclusion of a PhD dissertation is the last step of a long journey. However, the departure, the first step of the next journey, is the one which makes you think how much you have changed and what was the lesson of the previous journey. During these seven years in Amsterdam I have learned that "it is all about the crowd".

In this summary, of my eternal gratitude to the "crowd", I will not name anyone, with the exception of one person. A person who was my brother, my friend, my enemy, my father, my matte, my supervisor, my boss, and also my slave. Mr. Martin Kersten, also known as the fellow, the lunatic man, or some time ago, the guy with a mustache. He has shown that a mustache is what distinguishes a crazy scientist from a kid full of dreams. However, to succeed you must learn that good is enough.

The <famous_painter>DB team supported and protected me during the seven years I spent with him. A team composed by a great programmer and parttime "English dictionary"; the eagle eye; the SQL master with golden hands; the sailors, one on the ocean another on the cluster; the mafia people with a loud guy and a smart caveman, both commanded by the frappe girl. Bulgarian sweetness and an oriental touch were the ingredients for their balance often challenged by the "Vector ghetto", lead by the "Google geek" and the wellknown "You are wrong" superstar.

More people have left and entered the group over the years. To them a big thanks for telling me that more than getting a PhD, I should appreciate that I work at one of the best places in the Netherlands, CWI (Centrum Wiskunde & Informatica). The gadget boy from the Dacha, the Dacha Queen/Sergeant, and the cute blond girl who was always smiling, even on rainy days, were the ones that welcomed me to CWI. At the new wing of CWI, more people joined us and turned the group into a family. A family who succeeded in achieving an old dream of the Dacha's Queen, barbecues at MK's house.

Outside the office I never walked alone. Next to me was always someone from the famous Valckenierstraat, the amazing ISN (International Student Network), the Doos crowd, the glorious *Amsterdam gang*, dance floor queens and kings, random freaky people, an old lady from the supermarket, *birds*, and the most important ones, the lovely *Mokummers*.

Valckenierstraat people and ISN introduced me to the international life. Many were the ones I met at the borrel or in the garden with five couches to watch world cup games. With them I had wonderful house parties, nights at "Paradise, i.e., Paradiso", or shopping at "Free Market, i.e., Flea market". They showed me the streets of Amsterdam where several times "I was a loser, i.e., I was lost".

Few years later I witnessed the rise of the *Amsterdam gang*, also known as *I love you all*. They always made sure that the most painful hangovers were worth it. From India to Portugal, from Greece to Brazil, from Bulgaria to USA, they showed me that my culture is not better or worse than other cultures, it is just different. With this in mind, they wrote the most beautiful pages in my *book of life*. From trips abroad to dozens of dinners, from barbecues to hundreds of nights out, from several bike falls to thousands of liters of beer, they made me feel special, even when a taxi ran over me. Yes, tulip of Coimbra was right, *I love you all*.

So those, including the sweet and lovely snoopy, are the ones who made my life abroad as simple and sweet as back home, i.e., Balugães, place where my lovely family lives. Lately, snoopy gave it a magic touch. For whatever sweetness, joy of cooking, willingness to help people, and care for my guests I might have, you should thank my mother. For whatever strength, ambition, precision, word of honor, and motivation I might have, you should thank my father. If you think the combination of both personalities is not possible, then you should meet my two sisters. To them an enormous *obrigado* and a thousand apologies for not being there and showing how much I love you.

Thank you Amsterdam, it was an honor.

SIKS Dissertation Series

1998-1 Johan van den Akker (CWI) DEGAS - An Active, Temporal Database of Autonomous Objects

1998-2 Floris Wiesman (UM) Information Retrieval by Graphically Browsing Meta-Information

1998-3 Ans Steuten (TUD) A Contribution to the Linguistic Analysis of Business Conversations within the Language/Action Perspective

1998-4 Dennis Breuker (UM) Memory versus Search in Games

1998-5 E.W.Oskamp (RUL) Computerondersteuning bij Straftoemeting

1999-1 Mark Sloof (VU) Physiology of Quality Change Modelling; Automated modelling of Quality Change of Agricultural Products

1999-2 Rob Potharst (EUR) Classification using decision trees and neural nets

1999-3 Don Beal (UM) The Nature of Minimax Search

1999-4 Jacques Penders (UM) The practical Art of Moving Physical Objects

1999-5 Aldo de Moor (KUB) Empowering Communities: A Method for the Legitimate User-Driven Specification of Network Information Systems

1999-6 Niek J.E. Wijngaards (VU) Re-design of compositional systems

1999-7 David Spelt (UT) Verification support for object database design

1999-8 Jacques H.J. Lenting (UM) Informed Gambling: Conception and Analysis of a Multi-Agent Mechanism for Discrete Reallocation.

2000-1 Frank Niessink (VU) Perspectives on Improving Software Maintenance

2000-2 Koen Holtman (TUE) Prototyping of CMS Storage Management

2000-3 Carolien M.T. Metselaar (UVA) Sociaal-organisatorische gevolgen van kennistechnologie; een procesbenadering en actorperspectief.

2000-4 Geert de Haan (VU) ETAG, A Formal Model of Competence Knowledge for User Interface Design

2000-5 Ruud van der Pol (UM) Knowledge-based Query Formulation in Information Retrieval.

2000-6 Rogier van Eijk (UU) Programming Languages for Agent Communication

2000-7 Niels Peek (UU) Decision-theoretic Planning of Clinical Patient Management

2000-8 Veerle Coup (EUR) Sensitivity Analyis of Decision-Theoretic Networks

2000-9 Florian Waas (CWI) Principles of Probabilistic Query Optimization

2000-10 Niels Nes (CWI) Image Database Management System Design Considerations, Algorithms and Architecture

2000-11 Jonas Karlsson (CWI) Scalable Distributed Data Structures for Database Management

2001-1 Silja Renooij (UU) Qualitative Approaches to Quantifying Probabilistic Networks

2001-2 Koen Hindriks (UU) Agent Programming Languages: Programming with Mental Models

2001-3 Maarten van Someren (UvA) Learning as problem solving

2001-4 Evgueni Smirnov (UM) Conjunctive and Disjunctive Version Spaces with Instance-Based Boundary Sets

2001-5 Jacco van Ossenbruggen (VU) Processing Structured Hypermedia: A Matter of Style

2001-6 Martijn van Welie (VU) Task-based User Interface Design

2001-7 Bastiaan Schonhage (VU) Diva: Architectural Perspectives on Information Visualization

2001-8 Pascal van Eck (VU) A Compositional Semantic Structure for Multi-Agent Systems Dynamics.

2001-9 Pieter Jan 't Hoen (RUL) Towards Distributed Development of Large Object-Oriented Models, Views of Packages as Classes

2001-10 Maarten Sierhuis (UvA) Modeling and Simulating Work Practice BRAHMS: a multiagent modeling and simulation language for work practice analysis and design

2001-11 Tom M. van Engers (VUA) Knowledge Management: The Role of Mental Models in Business Systems Design

2002-01 Nico Lassing (VU) Architecture-Level Modifiability Analysis

2002-02 Roelof van Zwol (UT) Modelling and searching web-based document collections

2002-03 Henk Ernst Blok (UT) Database Optimization Aspects for Information Retrieval

2002-04 Juan Roberto Castelo Valdueza (UU) The Discrete Acyclic Digraph Markov Model in Data Mining

2002-05 Radu Serban (VU) The Private Cyberspace Modeling Electronic Environments inhabited by Privacy-concerned Agents

2002-06 Laurens Mommers (UL) Applied legal epistemology; Building a knowledge-based ontology of the legal domain

2002-07 Peter Boncz (CWI) Monet: A Next-Generation DBMS Kernel For Query-Intensive Applications

2002-08 Jaap Gordijn (VU) Value Based Requirements Engineering: Exploring Innovative E-Commerce Ideas

2002-09 Willem-Jan van den Heuvel(KUB) Integrating Modern Business Applications with Objectified Legacy Systems

2002-10 Brian Sheppard (UM) Towards Perfect Play of Scrabble

2002-11 Wouter C.A. Wijngaards (VU) Agent Based Modelling of Dynamics: Biological and Organisational Applications

2002-12 Albrecht Schmidt (Uva) Processing XML in Database Systems

2002-13 Hongjing Wu (TUE) A Reference Architecture for Adaptive Hypermedia Applications

2002-14 Wieke de Vries (UU) Agent Interaction: Abstract Approaches to Modelling, Programming and Verifying Multi-Agent Systems

2002-15 Rik Eshuis (UT) Semantics and Verification of UML Activity Diagrams for Workflow Modelling

2002-16 Pieter van Langen (VU) The Anatomy of Design: Foundations, Models and Applications

2002-17 Stefan Manegold (UVA) Understanding, Modeling, and Improving Main-Memory Database Performance

2003-01 Heiner Stuckenschmidt (VU) Ontology-Based Information Sharing in Weakly Structured Environments

2003-02 Jan Broersen (VU) Modal Action Logics for Reasoning About Reactive Systems

2003-03 Martijn Schuemie (TUD) Human-Computer Interaction and Presence in Virtual Reality Exposure Therapy

2003-04 Milan Petkovic (UT) Content-Based Video Retrieval Supported by Database Technology

2003-05 Jos Lehmann (UVA) Causation in Artificial Intelligence and Law - A modelling approach

2003-06 Boris van Schooten (UT) Development and specification of virtual environments

2003-07 Machiel Jansen (UvA) Formal Explorations of Knowledge Intensive Tasks

2003-08 Yongping Ran (UM) Repair Based Scheduling

2003-09 Rens Kortmann (UM) The resolution of visually guided behaviour

2003-10 Andreas Lincke (UvT) Electronic Business Negotiation: Some experimental studies on the interaction between medium, innovation context and culture

2003-11 Simon Keizer (UT) Reasoning under Uncertainty in Natural Language Dialogue using Bayesian Networks

2003-12 Roeland Ordelman (UT) Dutch speech recognition in multimedia information retrieval

2003-13 Jeroen Donkers (UM) Nosce Hostem - Searching with Opponent Models

2003-14 Stijn Hoppenbrouwers (KUN) Freezing Language: Conceptualisation Processes across ICT-Supported Organisations

2003-15 Mathijs de Weerdt (TUD) Plan Merging in Multi-Agent Systems

2003-16 Menzo Windhouwer (CWI) Feature Grammar Systems - Incremental Maintenance of Indexes to Digital Media Warehouses

2003-17 David Jansen (UT) Extensions of Statecharts with Probability, Time, and Stochastic Timing

2003-18 Levente Kocsis (UM) Learning Search Decisions

2004-01 Virginia Dignum (UU) A Model for Organizational Interaction: Based on Agents, Founded in Logic

2004-02 Lai Xu (UvT) Monitoring Multi-party Contracts for E-business

2004-03 Perry Groot (VU) A Theoretical and Empirical Analysis of Approximation in Symbolic Problem Solving

2004-04 Chris van Aart (UVA) Organizational Principles for Multi-Agent Architectures

2004-05 Viara Popova (EUR) Knowledge discovery and monotonicity

2004-06 Bart-Jan Hommes (TUD) The Evaluation of Business Process Modeling Techniques

2004-07 Elise Boltjes (UM) Voorbeeldig onderwijs; voorbeeldgestuurd onderwijs, een opstap naar abstract denken, vooral voor meisjes

2004-08 Joop Verbeek(UM) Politie en de Nieuwe Internationale Informatiemarkt, Grensregionale politiële gegevensuitwisseling en digitale expertise

2004-09 Martin Caminada (VU) For the Sake of the Argument; explorations into argument-based reasoning

2004-10 Suzanne Kabel (UVA) Knowledge-rich indexing of learning-objects

2004-11 Michel Klein (VU) Change Management for Distributed Ontologies

2004-12 The Duy Bui (UT) Creating emotions and facial expressions for embodied agents

2004-13 Wojciech Jamroga (UT) Using Multiple Models of Reality: On Agents who Know how to Play

2004-14 Paul Harrenstein (UU) Logic in Conflict. Logical Explorations in Strategic Equilibrium

2004-15 Arno Knobbe (UU) Multi-Relational Data Mining

2004-16 Federico Divina (VU) Hybrid Genetic Relational Search for Inductive Learning

2004-17 Mark Winands (UM) Informed Search in Complex Games

2004-18 Vania Bessa Machado (UvA) Supporting the Construction of Qualitative Knowledge Models

2004-19 Thijs Westerveld (UT) Using generative probabilistic models for multimedia retrieval

2004-20 Madelon Evers (Nyenrode) Learning from Design: facilitating multidisciplinary design teams

2005-01 Floor Verdenius (UVA) Methodological Aspects of Designing Induction-Based Applications

2005-02 Erik van der Werf (UM) AI techniques for the game of Go

2005-03 Franc Grootjen (RUN) A Pragmatic Approach to the Conceptualisation of Language

2005-04 Nirvana Meratnia (UT) Towards Database Support for Moving Object data

2005-05 Gabriel Infante-Lopez (UVA) Two-Level Probabilistic Grammars for Natural Language Parsing

2005-06 Pieter Spronck (UM) Adaptive Game AI

2005-07 Flavius Frasincar (TUE) Hypermedia Presentation Generation for Semantic Web Information Systems

2005-08 Richard Vdovjak (TUE) A Model-driven Approach for Building Distributed Ontology-based Web Applications

2005-09 Jeen Broekstra (VU) Storage, Querying and Inferencing for Semantic Web Languages

2005-10 Anders Bouwer (UVA) Explaining Behaviour: Using Qualitative Simulation in Interactive Learning Environments

2005-11 Elth Ogston (VU) Agent Based Matchmaking and Clustering - A Decentralized Approach to Search

2005-12 Csaba Boer (EUR) Distributed Simulation in Industry

2005-13 Fred Hamburg (UL) Een Computermodel voor het Ondersteunen van Euthanasiebeslissingen

2005-14 Borys Omelayenko (VU) Web-Service configuration on the Semantic Web; Exploring how semantics meets pragmatics

2005-15 Tibor Bosse (VU) Analysis of the Dynamics of Cognitive Processes

2005-16 Joris Graaumans (UU) Usability of XML Query Languages

2005-17 Boris Shishkov (TUD) Software Specification Based on Re-usable Business Components

2005-18 Danielle Sent (UU) Test-selection strategies for probabilistic networks

2005-19 Michel van Dartel (UM) Situated Representation

2005-20 Cristina Coteanu (UL) Cyber Consumer Law, State of the Art and Perspectives

2005-21 Wijnand Derks (UT) Improving Concurrency and Recovery in Database Systems by Exploiting Application Semantics

2006-01 Samuil Angelov (TUE) Foundations of B2B Electronic Contracting

2006-02 Cristina Chisalita (VU) Contextual issues in the design and use of information technology in organizations

2006-03 Noor Christoph (UVA) The role of metacognitive skills in learning to solve problems

2006-04 Marta Sabou (VU) Building Web Service Ontologies

2006-05 Cees Pierik (UU) Validation Techniques for Object-Oriented Proof Outlines

2006-06 Ziv Baida (VU) Software-aided Service Bundling - Intelligent Methods & Tools for Graphical Service Modeling

2006-07 Marko Smiljanic (UT) XML schema matching – balancing efficiency and effectiveness by means of clustering

2006-08 Eelco Herder (UT) Forward, Back and Home Again - Analyzing User Behavior on the Web

2006-09 Mohamed Wahdan (UM) Automatic Formulation of the Auditor's Opinion

2006-10 Ronny Siebes (VU) Semantic Routing in Peer-to-Peer Systems

2006-11 Joeri van Ruth (UT) Flattening Queries over Nested Data Types

2006-12 Bert Bongers (VU) Interactivation - Towards an e-cology of people, our technological environment, and the arts

2006-13 Henk-Jan Lebbink (UU) Dialogue and Decision Games for Information Exchanging Agents

2006-14 Johan Hoorn (VU) Software Requirements: Update, Upgrade, Redesign - towards a Theory of Requirements Change

2006-15 Rainer Malik (UU) CONAN: Text Mining in the Biomedical Domain

2006-16 Carsten Riggelsen (UU) Approximation Methods for Efficient Learning of Bayesian Networks

2006-17 Stacey Nagata (UU) User Assistance for Multitasking with Interruptions on a Mobile Device

2006-18 Valentin Zhizhkun (UVA) Graph transformation for Natural Language Processing

2006-19 Birna van Riemsdijk (UU) Cognitive Agent Programming: A Semantic Approach

2006-20 Marina Velikova (UvT) Monotone models for prediction in data mining

2006-21 Bas van Gils (RUN) Aptness on the Web

2006-22 Paul de Vrieze (RUN) Fundaments of Adaptive Personalisation

2006-23 Ion Juvina (UU) Development of Cognitive Model for Navigating on the Web

2006-24 Laura Hollink (VU) Semantic Annotation for Retrieval of Visual Resources

2006-25 Madalina Drugan (UU) Conditional log-likelihood MDL and Evolutionary MCMC

2006-26 Vojkan Mihajlovi'c (UT) Score Region Algebra: A Flexible Framework for Structured Information Retrieval

2006-27 Stefano Bocconi (CWI) Vox Populi: generating video documentaries from semantically annotated media repositories

2006-28 Borkur Sigurbjornsson (UVA) Focused Information Access using XML Element Retrieval

2007-01 Kees Leune (UvT) Access Control and Service-Oriented Architectures

2007-02 Wouter Teepe (RUG) Reconciling Information Exchange and Confidentiality: A Formal Approach

2007-03 Peter Mika (VU) Social Networks and the Semantic Web

2007-04 Jurriaan van Diggelen (UU) Achieving Semantic Interoperability in Multi-agent Systems: a dialogue-based approach

2007-05 Bart Schermer (UL) Software Agents, Surveillance, and the Right to Privacy: a Legislative Framework for Agent-enabled Surveillance

2007-06 Gilad Mishne (UVA) Applied Text Analytics for Blogs

2007-07 Natasa Jovanovic' (UT) To Whom It May Concern - Addressee Identification in Face-to-Face Meetings

2007-08 Mark Hoogendoorn (VU) Modeling of Change in Multi-Agent Organizations

2007-09 David Mobach (VU) Agent-Based Mediated Service Negotiation

2007-10 Huib Aldewereld (UU) Autonomy vs. Conformity: an Institutional Perspective on Norms and Protocols

2007-11 Natalia Stash (TUE) Incorporating Cognitive/Learning Styles in a General-Purpose Adaptive Hypermedia System

2007-12 Marcel van Gerven (RUN) Bayesian Networks for Clinical Decision Support: A Rational Approach to Dynamic Decision-Making under Uncertainty

2007-13 Rutger Rienks (UT) Meetings in Smart Environments; Implications of Progressing Technology

2007-14 Niek Bergboer (UM) Context-Based Image Analysis

2007-15 Joyca Lacroix (UM) NIM: a Situated Computational Memory Model

2007-16 Davide Grossi (UU) Designing Invisible Handcuffs. Formal investigations in Institutions and Organizations for Multi-agent Systems

2007-17 Theodore Charitos (UU) Reasoning with Dynamic Networks in Practice

2007-18 Bart Orriens (UvT) On the development an management of adaptive business collaborations

2007-19 David Levy (UM) Intimate relationships with artificial partners

2007-20 Slinger Jansen (UU) Customer Configuration Updating in a Software Supply Network

2007-21 Karianne Vermaas (UU) Fast diffusion and broadening use: A research on residential adoption and usage of broadband internet in the Netherlands between 2001 and 2005

2007-22 Zlatko Zlatev (UT) Goal-oriented design of value and process models from patterns

2007-23 Peter Barna (TUE) Specification of Application Logic in Web Information Systems

2007-24 Georgina Ramrez Camps (CWI) Structural Features in XML Retrieval

2007-25 Joost Schalken (VU) Empirical Investigations in Software Process Improvement

2008-01 Katalin Boer-Sorbn (EUR) Agent-Based Simulation of Financial Markets: A modular,continuous-time approach

2008-02 Alexei Sharpanskykh (VU) On Computer-Aided Methods for Modeling and Analysis of Organizations

2008-03 Vera Hollink (UVA) Optimizing hierarchical menus: a usage-based approach

2008-04 Ander de Keijzer (UT) Management of Uncertain Data - towards unattended integration

2008-05 Bela Mutschler (UT) Modeling and simulating causal dependencies on process-aware information systems from a cost perspective

2008-06 Arjen Hommersom (RUN) On the Application of Formal Methods to Clinical Guidelines, an Artificial Intelligence Perspective

2008-07 Peter van Rosmalen (OU) Supporting the tutor in the design and support of adaptive e-learning

2008-08 Janneke Bolt (UU) Bayesian Networks: Aspects of Approximate Inference

2008-09 Christof van Nimwegen (UU) The paradox of the guided user: assistance can be counter-effective

2008-10 Wauter Bosma (UT) Discourse oriented summarization

2008-11 Vera Kartseva (VU) Designing Controls for Network Organizations: A Value-Based Approach

2008-12 Jozsef Farkas (RUN) A Semiotically Oriented Cognitive Model of Knowledge Representation

2008-13 Caterina Carraciolo (UVA) Topic Driven Access to Scientific Handbooks

2008-14 Arthur van Bunningen (UT) Context-Aware Querying; Better Answers with Less Effort

2008-15 Martijn van Otterlo (UT) The Logic of Adaptive Behavior: Knowledge Representation and Algorithms for the Markov Decision Process Framework in First-Order Domains.

2008-16 Henriette van Vugt (VU) Embodied agents from a user's perspective

2008-17 Martin Op 't Land (TUD) Applying Architecture and Ontology to the Splitting and Allying of Enterprises

2008-18 Guido de Croon (UM) Adaptive Active Vision

2008-19 Henning Rode (UT) From Document to Entity Retrieval: Improving Precision and Performance of Focused Text Search

2008-20 Rex Arendsen (UVA) Geen bericht, goed bericht. Een onderzoek naar de effecten van de introductie van elektronisch berichtenverkeer met de overheid op de administratieve lasten van bedrijven

2008-21 Krisztian Balog (UVA) People Search in the Enterprise

2008-22 Henk Koning (UU) Communication of IT-Architecture

2008-23 Stefan Visscher (UU) Bayesian network models for the management of ventilator-associated pneumonia

2008-24 Zharko Aleksovski (VU) Using background knowledge in ontology matching

2008-25 Geert Jonker (UU) Efficient and Equitable Exchange in Air Traffic Management Plan Repair using Spender-signed Currency

2008-26 Marijn Huijbregts (UT) Segmentation, Diarization and Speech Transcription: Surprise Data Unraveled

2008-27 Hubert Vogten (OU) Design and Implementation Strategies for IMS Learning Design

2008-28 Ildiko Flesch (RUN) On the Use of Independence Relations in Bayesian Networks

2008-29 Dennis Reidsma (UT) Annotations and Subjective Machines - Of Annotators, Embodied Agents, Users, and Other Humans

2008-30 Wouter van Atteveldt (VU) Semantic Network Analysis: Techniques for Extracting, Representing and Querying Media Content

2008-31 Loes Braun (UM) Pro-Active Medical Information Retrieval

2008-32 Trung H. Bui (UT) Toward Affective Dialogue Management using Partially Observable Markov Decision Processes

2008-33 Frank Terpstra (UVA) Scientific Workflow Design; theoretical and practical issues

2008-34 Jeroen de Knijf (UU) Studies in Frequent Tree Mining

2008-35 Ben Torben Nielsen (UvT) Dendritic morphologies: function shapes structure

2009-01 Rasa Jurgelenaite (RUN) Symmetric Causal Independence Models

2009-02 Willem Robert van Hage (VU) Evaluating Ontology-Alignment Techniques

2009-03 Hans Stol (UvT) A Framework for Evidence-based Policy Making Using IT

2009-04 Josephine Nabukenya (RUN) Improving the Quality of Organisational Policy Making using Collaboration Engineering

2009-05 Sietse Overbeek (RUN) Bridging Supply and Demand for Knowledge Intensive Tasks - Based on Knowledge, Cognition, and Quality

2009-06 Muhammad Subianto (UU) Understanding Classification

2009-07 Ronald Poppe (UT) Discriminative Vision-Based Recovery and Recognition of Human Motion

2009-08 Volker Nannen (VU) Evolutionary Agent-Based Policy Analysis in Dynamic Environments

2009-09 Benjamin Kanagwa (RUN) Design, Discovery and Construction of Service-oriented Systems

2009-10 Jan Wielemaker (UVA) Logic programming for knowledge-intensive interactive applications

2009-11 Alexander Boer (UVA) Legal Theory, Sources of Law & the Semantic Web

2009-12 Peter Massuthe (TUE, Humboldt-Universitaet zu Berlin) Operating Guidelines for Services

2009-13 Steven de Jong (UM) Fairness in Multi-Agent Systems

2009-14 Maksym Korotkiy (VU) From ontology-enabled services to service-enabled ontologies (making ontologies work in e-science with ONTO-SOA)

2009-15 Rinke Hoekstra (UVA) Ontology Representation - Design Patterns and Ontologies that Make Sense

2009-16 Fritz Reul (UvT) New Architectures in Computer Chess

2009-17 Laurens van der Maaten (UvT) Feature Extraction from Visual Data

2009-18 Fabian Groffen (CWI) Armada, An Evolving Database System

2009-19 Valentin Robu (CWI) Modeling Preferences, Strategic Reasoning and Collaboration in Agent-Mediated Electronic Markets

2009-20 Bob van der Vecht (UU) Adjustable Autonomy: Controling Influences on Decision Making

2009-21 Stijn Vanderlooy (UM) Ranking and Reliable Classification

2009-22 Pavel Serdyukov (UT) Search For Expertise: Going beyond direct evidence

2009-23 Peter Hofgesang (VU) Modelling Web Usage in a Changing Environment

2009-24 Annerieke Heuvelink (VUA) Cognitive Models for Training Simulations

2009-25 Alex van Ballegooij (CWI) "RAM: Array Database Management through Relational Mapping"

2009-26 Fernando Koch (UU) An Agent-Based Model for the Development of Intelligent Mobile Services

2009-27 Christian Glahn (OU) Contextual Support of social Engagement and Reflection on the Web

2009-28 Sander Evers (UT) Sensor Data Management with Probabilistic Models

2009-29 Stanislav Pokraev (UT) Model-Driven Semantic Integration of Service-Oriented Applications

2009-30 Marcin Zukowski (CWI) Balancing vectorized query execution with bandwidth-optimized storage

2009-31 Sofiya Katrenko (UVA) A Closer Look at Learning Relations from Text

2009-32 Rik Farenhorst (VU) and Remco de Boer (VU) Architectural Knowledge Management: Supporting Architects and Auditors

2009-33 Khiet Truong (UT) How Does Real Affect Affect Affect Recognition In Speech?

2009-34 Inge van de Weerd (UU) Advancing in Software Product Management: An Incremental Method Engineering Approach

2009-35 Wouter Koelewijn (UL) Privacy en Politiegegevens; Over geautomatiseerde normatieve informatie-uitwisseling

2009-36 Marco Kalz (OUN) Placement Support for Learners in Learning Networks

2009-37 Hendrik Drachsler (OUN) Navigation Support for Learners in Informal Learning Networks

2009-38 Riina Vuorikari (OU) Tags and self-organisation: a metadata ecology for learning resources in a multilingual context

2009-39 Christian Stahl (TUE, Humboldt-Universitaet zu Berlin) Service Substitution – A Behavioral Approach Based on Petri Nets

2009-40 Stephan Raaijmakers (UvT) Multinomial Language Learning: Investigations into the Geometry of Language

2009-41 Igor Berezhnyy (UvT) Digital Analysis of Paintings

2009-42 Toine Bogers Recommender Systems for Social Bookmarking

2009-43 Virginia Nunes Leal Franqueira (UT) Finding Multi-step Attacks in Computer Networks using Heuristic Search and Mobile Ambients

2009-44 Roberto Santana Tapia (UT) Assessing Business-IT Alignment in Networked Organizations

2009-45 Jilles Vreeken (UU) Making Pattern Mining Useful

2009-46 Loredana Afanasiev (UvA) Querying XML: Benchmarks and Recursion

2010-01 Matthijs van Leeuwen (UU) Patterns that Matter

2010-02 Ingo Wassink (UT) Work flows in Life Science

2010-03 Joost Geurts (CWI) A Document Engineering Model and Processing Framework for Multimedia documents

2010-04 Olga Kulyk (UT) Do You Know What I Know? Situational Awareness of Co-located Teams in Multidisplay Environments

2010-05 Claudia Hauff (UT) Predicting the Effectiveness of Queries and Retrieval Systems

2010-06 Sander Bakkes (UvT) Rapid Adaptation of Video Game AI

2010-07 Wim Fikkert (UT) Gesture interaction at a Distance

2010-08 Krzysztof Siewicz (UL) Towards an Improved Regulatory Framework of Free Software. Protecting user freedoms in a world of software communities and eGovernments

2010-09 Hugo Kielman (UL) A Politiele gegevensverwerking en Privacy, Naar een effectieve waarborging

2010-10 Rebecca Ong (UL) Mobile Communication and Protection of Children

2010-11 Adriaan Ter Mors (TUD) The world according to MARP: Multi-Agent Route Planning

2010-12 Susan van den Braak (UU) Sensemaking software for crime analysis

2010-13 Gianluigi Folino (RUN) High Performance Data Mining using Bio-inspired techniques

2010-14 Sander van Splunter (VU) Automated Web Service Reconfiguration

2010-15 Lianne Bodenstaff (UT) Managing Dependency Relations in Inter-Organizational Models

2010-16 Sicco Verwer (TUD) Efficient Identification of Timed Automata, theory and practice

2010-17 Spyros Kotoulas (VU) Scalable Discovery of Networked Resources: Algorithms, Infrastructure, Applications

2010-18 Charlotte Gerritsen (VU) Caught in the Act: Investigating Crime by Agent-Based Simulation

2010-19 Henriette Cramer (UvA) People's Responses to Autonomous and Adaptive Systems

2010-20 Ivo Swartjes (UT) Whose Story Is It Anyway? How Improv Informs Agency and Authorship of Emergent Narrative

2010-21 Harold van Heerde (UT) Privacy-aware data management by means of data degradation

2010-22 Michiel Hildebrand (CWI) End-user Support for Access to Heterogeneous Linked Data

2010-23 Bas Steunebrink (UU) The Logical Structure of Emotions

2010-24 Dmytro Tykhonov Designing Generic and Efficient Negotiation Strategies

2010-25 Zulfiqar Ali Memon (VU) Modelling Human-Awareness for Ambient Agents: A Human Mindreading Perspective

2010-26 Ying Zhang (CWI) XRPC: Efficient Distributed Query Processing on Heterogeneous XQuery Engines

2010-27 Marten Voulon (UL) Automatisch contracteren

2010-28 Arne Koopman (UU) Characteristic Relational Patterns

2010-29 Stratos Idreos(CWI) Database Cracking: Towards Auto-tuning Database Kernels

2010-30 Marieke van Erp (UvT) Accessing Natural History - Discoveries in data cleaning, structuring, and retrieval

2010-31 Victor de Boer (UVA) Ontology Enrichment from Heterogeneous Sources on the Web

2010-32 Marcel Hiel (UvT) An Adaptive Service Oriented Architecture: Automatically solving Interoperability Problems

2010-33 Robin Aly (UT) Modeling Representation Uncertainty in Concept-Based Multimedia Retrieval

2010-34 Teduh Dirgahayu (UT) Interaction Design in Service Compositions

2010-35 Dolf Trieschnigg (UT) Proof of Concept: Concept-based Biomedical Information Retrieval

2010-36 Jose Janssen (OU) Paving the Way for Lifelong Learning; Facilitating competence development through a learning path specification

2010-37 Niels Lohmann (TUE) Correctness of services and their composition

2010-38 Dirk Fahland (TUE) From Scenarios to components

2010-39 Ghazanfar Farooq Siddiqui (VU) Integrative modeling of emotions in virtual agents

2010-40 Mark van Assem (VU) Converting and Integrating Vocabularies for the Semantic Web

2010-41 Guillaume Chaslot (UM) Monte-Carlo Tree Search

2010-42 Sybren de Kinderen (VU) Needs-driven service bundling in a multi-supplier setting - the computational e3-service approach

2010-43 Peter van Kranenburg (UU) A Computational Approach to Content-Based Retrieval of Folk Song Melodies

2010-44 Pieter Bellekens (TUE) An Approach towards Context-sensitive and User-adapted Access to Heterogeneous Data Sources, Illustrated in the Television Domain

2010-45 Vasilios Andrikopoulos (UvT) A theory and model for the evolution of software services

2010-46 Vincent Pijpers (VU) e3alignment: Exploring Inter-Organizational Business-ICT Alignment

2010-47 Chen Li (UT) Mining Process Model Variants: Challenges, Techniques, Examples

2010-48 Milan Lovric (EUR) Behavioral Finance and Agent-Based Artificial Markets

2010-49 Jahn-Takeshi Saito (UM) Solving difficult game positions

2010-50 Bouke Huurnink (UVA) Search in Audiovisual Broadcast Archives

2010-51 Alia Khairia Amin (CWI) Understanding and supporting information seeking tasks in multiple sources

2010-52 Peter-Paul van Maanen (VU) Adaptive Support for Human-Computer Teams: Exploring the Use of Cognitive Models of Trust and Attention

2010-53 Edgar Meij (UVA) Combining Concepts and Language Models for Information Access

2011-01 Botond Cseke (RUN) Variational Algorithms for Bayesian Inference in Latent Gaussian Models

2011-02 Nick Tinnemeier(UU) Organizing Agent Organizations. Syntax and Operational Semantics of an Organization-Oriented Programming Language

2011-03 Jan Martijn van der Werf (TUE) Compositional Design and Verification of Component-Based Information Systems

2011-04 Hado van Hasselt (UU) Insights in Reinforcement Learning; Formal analysis and empirical evaluation of temporal-difference learning algorithms

2011-05 Base van der Raadt (VU) Enterprise Architecture Coming of Age - Increasing the Performance of an Emerging Discipline.

2011-06 Yiwen Wang (TUE) Semantically-Enhanced Recommendations in Cultural Heritage

2011-07 Yujia Cao (UT) Multimodal Information Presentation for High Load Human Computer Interaction

2011-08 Nieske Vergunst (UU) BDI-based Generation of Robust Task-Oriented Dialogues

2011-09 Tim de Jong (OU) Contextualised Mobile Media for Learning

2011-10 Bart Bogaert (UvT) Cloud Content Contention

2011-11 Dhaval Vyas (UT) Designing for Awareness: An Experience-focused HCI Perspective

2011-12 Carmen Bratosin (TUE) Grid Architecture for Distributed Process Mining

2011-13 Xiaoyu Mao (UvT) Airport under Control. Multiagent Scheduling for Airport Ground Handling

2011-14 Milan Lovric (EUR) Behavioral Finance and Agent-Based Artificial Markets

2011-15 Marijn Koolen (UvA) The Meaning of Structure: the Value of Link Evidence for Information Retrieval

2011-16 Maarten Schadd (UM) Selective Search in Games of Different Complexity

2011-17 Jiyin He (UVA) Exploring Topic Structure: Coherence, Diversity and Relatedness

2011-18 Mark Ponsen (UM) Strategic Decision-Making in complex games

2011-19 Ellen Rusman (OU) The Mind ' s Eye on Personal Profiles

2011-20 Qing Gu (VU) Guiding service-oriented software engineering - A view-based approach

2011-21 Linda Terlouw (TUD) Modularization and Specification of Service-Oriented Systems

2011-22 Junte Zhang (UVA) System Evaluation of Archival Description and Access

2011-23 Wouter Weerkamp (UVA) Finding People and their Utterances in Social Media

2011-24 Herwin van Welbergen (UT) Behavior Generation for Interpersonal Coordination with Virtual Humans On Specifying, Scheduling and Realizing Multimodal Virtual Human Behavior

2011-25 Syed Waqar ul Qounain Jaffry (VU) Analysis and Validation of Models for Trust Dynamics

2011-26 Matthijs Aart Pontier (VU) Virtual Agents for Human Communication - Emotion Regulation and Involvement-Distance Trade-Offs in Embodied Conversational Agents and Robots

2011-27 Aniel Bhulai (VU) Dynamic website optimization through autonomous management of design patterns

2011-28 Rianne Kaptein(UVA) Effective Focused Retrieval by Exploiting Query Context and Document Structure

2011-29 Faisal Kamiran (TUE) Discrimination-aware Classification

2011-30 Egon van den Broek (UT) Affective Signal Processing (ASP): Unraveling the mystery of emotions

2011-31 Ludo Waltman (EUR) Computational and Game-Theoretic Approaches for Modeling Bounded Rationality

2011-32 Nees-Jan van Eck (EUR) Methodological Advances in Bibliometric Mapping of Science

2011-33 Tom van der Weide (UU) Arguing to Motivate Decisions

2011-34 Paolo Turrini (UU) Strategic Reasoning in Interdependence: Logical and Game-theoretical Investigations

2011-35 Maaike Harbers (UU) Explaining Agent Behavior in Virtual Training

2011-36 Erik van der Spek (UU) Experiments in serious game design: a cognitive approach

2011-37 Adriana Burlutiu (RUN) Machine Learning for Pairwise Data, Applications for Preference Learning and Supervised Network Inference

2011-38 Nyree Lemmens (UM) Bee-inspired Distributed Optimization

2011-39 Joost Westra (UU) Organizing Adaptation using Agents in Serious Games

2011-40 Viktor Clerc (VU) Architectural Knowledge Management in Global Software Development

2011-41 Luan Ibraimi (UT) Cryptographically Enforced Distributed Data Access Control

2011-42 Michal Sindlar (UU) Explaining Behavior through Mental State Attribution

2011-43 Henk van der Schuur (UU) Process Improvement through Software Operation Knowledge

2011-44 Boris Reuderink (UT) Robust Brain-Computer Interfaces

2011-45 Herman Stehouwer (UvT) Statistical Language Models for Alternative Sequence Selection

2011-46 Beibei Hu (TUD) Towards Contextualized Information Delivery: A Rule-based Architecture for the Domain of Mobile Police Work

2011-47 Azizi Bin Ab Aziz(VU) Exploring Computational Models for Intelligent Support of Persons with Depression

2011-48 Mark Ter Maat (UT) Response Selection and Turn-taking for a Sensitive Artificial Listening Agent

2011-49 Andreea Niculescu (UT) Conversational interfaces for task-oriented spoken dialogues: design aspects influencing interaction quality

2012-01 Terry Kakeeto (UvT) Relationship Marketing for SMEs in Uganda

2012-02 Muhammad Umair(VU) Adaptivity, emotion, and Rationality in Human and Ambient Agent Models

2012-03 Adam Vanya (VU) Supporting Architecture Evolution by Mining Software Repositories

2012-04 Jurriaan Souer (UU) Development of Content Management System-based Web Applications

2012-05 Marijn Plomp (UU) Maturing Interorganisational Information Systems

2012-06 Wolfgang Reinhardt (OU) Awareness Support for Knowledge Workers in Research Networks

2012-07 Rianne van Lambalgen (VU) When the Going Gets Tough: Exploring Agent-based Models of Human Performance under Demanding Conditions

2012-08 Gerben de Vries (UVA) Kernel Methods for Vessel Trajectories

2012-09 Ricardo Neisse (UT) Trust and Privacy Management Support for Context-Aware Service Platforms

2012-10 David Smits (TUE) Towards a Generic Distributed Adaptive Hypermedia Environment

2012-11 J.C.B. Rantham Prabhakara (TUE) Process Mining in the Large: Preprocessing, Discovery, and Diagnostics

2012-12 Kees van der Sluijs (TUE) Model Driven Design and Data Integration in Semantic Web Information Systems

2012-13 Suleman Shahid (UvT) Fun and Face: Exploring non-verbal expressions of emotion during playful interactions

2012-14 Evgeny Knutov(TUE) Generic Adaptation Framework for Unifying Adaptive Web-based Systems

2012-15 Natalie van der Wal (VU) Social Agents. Agent-Based Modelling of Integrated Internal and Social Dynamics of Cognitive and Affective Processes

2012-16 Fiemke Both (VU) Helping people by understanding them - Ambient Agents supporting task execution and depression treatment

2012-17 Amal Elgammal (UvT) Towards a Comprehensive Framework for Business Process Compliance

2012-18 Eltjo Poort (VU) Improving Solution Architecting Practices

2012-19 Helen Schonenberg (TUE) What's Next? Operational Support for Business Process Execution

2012-20 Ali Bahramisharif (RUN) Covert Visual Spatial Attention, a Robust Paradigm for Brain-Computer Interfacing

2012-21 Roberto Cornacchia (TUD) Querying Sparse Matrices for Information Retrieval

2012-22 Thijs Vis (UvT) Intelligence, politie en veiligheidsdienst: verenigbare grootheden?

2012-23 Christian Muehl (UT) Toward Affective Brain-Computer Interfaces: Exploring the Neurophysiology of Affect during Human Media Interaction

2012-24 Laurens van der Werff (UT) Evaluation of Noisy Transcripts for Spoken Document Retrieval

2012-25 Silja Eckartz (UT) Managing the Business Case Development in Inter-Organizational IT Projects: A Methodology and its Application

2012-26 Emile de Maat (UVA) Making Sense of Legal Text

2012-27 Hayrettin Gürkök (UT) Mind the Sheep! User Experience Evaluation & Brain-Computer Interface Games

2012-28 Nancy Pascall (UvT) Engendering Technology Empowering Women

2012-29 Almer Tigelaar (UT) Peer-to-Peer Information Retrieval

2012-30 Alina Pommeranz (TUD) Designing Human-Centered Systems for Reflective Decision Making

2012-31 Emily Bagarukayo (RUN) A Learning by Construction Approach for Higher Order Cognitive Skills Improvement, Building Capacity and Infrastructure

2012-32 Wietske Visser (TUD) Qualitative multi-criteria preference representation and reasoning

2012-33 Rory Sie (OUN) Coalitions in Cooperation Networks (COCOON)

2012-34 Pavol Jancura (RUN) Evolutionary analysis in PPI networks and applications

2012-35 Evert Haasdijk (VU) Never Too Old To Learn – On-line Evolution of Controllers in Swarm- and Modular Robotics

2012-36 Denis Ssebugwawo (RUN) Analysis and Evaluation of Collaborative Modeling Processes

2012-37 Agnes Nakakawa (RUN) A Collaboration Process for Enterprise Architecture Creation

2012-38 Selmar Smit (VU) Parameter Tuning and Scientific Testing in Evolutionary Algorithms

2012-39 Hassan Fatemi (UT) Risk-aware design of value and coordination networks

2012-40 Agus Gunawan (UvT) Information Access for SMEs in Indonesia

2012-41 Sebastian Kelle (OU) Game Design Patterns for Learning

2012-42 Dominique Verpoorten (OU) Reflection Amplifiers in self-regulated Learning

2012-44 Anna Tordai (VU) On Combining Alignment Techniques

2012-45 Benedikt Kratz (UvT) A Model and Language for Business-aware Transactions

2012-46 Simon Carter (UVA) Exploration and Exploitation of Multilingual Data for Statistical Machine Translation

2012-47 Manos Tsagkias (UVA) Mining Social Media: Tracking Content and Predicting Behavior

2012-48 Jorn Bakker (TUE) Handling Abrupt Changes in Evolving Time-series Data

2012-49 Michael Kaisers (UM) Learning against Learning - Evolutionary dynamics of reinforcement learning algorithms in strategic interactions

2013-01 Viorel Milea (EUR) News Analytics for Financial Decision Support

2013-02 Erietta Liarou (CWI) MonetDB/DataCell: Leveraging the Column-store Database Technology for Efficient and Scalable Stream Processing

2013-03 Szymon Klarman (VU) Reasoning with Contexts in Description Logics

2013-04 Chetan Yadati (TUD) Coordinating autonomous planning and scheduling

2013-05 Dulce Pumareja (UT) Groupware Requirements Evolutions Patterns