Turning Scientists into Data Explorers

Yağız Kargın
CWI Amsterdam
yagiz.kargin@cwi.nl
Expected Graduation Date: 2016
Supervised by Martin Kersten

ABSTRACT

Nowadays scientists receive increasingly large volumes of data daily. These volumes and accompanying metadata that describes them are collected in scientific file repositories. Today's scientists need a data management tool that makes these file repositories accessible and performs a number of exploration steps near-instantly. Current database technology, however, has a long *data-to-insight* time, and does not provide enough interactivity to shorten the exploration time. We envision that exploiting metadata helps solving these problems. To this end, we propose a novel query execution paradigm, in which we decompose the query execution into two stages. During the first stage, we process only metadata, whereas the rest of the data is processed during the second stage. So that, we can exploit metadata to boost interactivity and to ingest only required data per query transparently. Preliminary experiments show that up-front ingestion time is reduced by orders of magnitude, while query performance remains similar. Motivated by these results, we identify the challenges on the way from the new paradigm to efficient interactive data exploration.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*Query Processing*; H.2.8 [**Database Applications**]: Scientific Databases

General Terms

Design, Performance, Algorithms

Keywords

Two-stage Query Execution; Data Exploration; Scientific Data

1. INTRODUCTION

In the old pen-and-paper days, scientists used to have full insight and control over their data. In the e-science era, it is foreseen that they automatically receive multiple terabytes of data on a daily basis. Since this data is not generated through their own observations and/or it is simply too large, (*i*) it is not natural for them anymore to maintain the insight, they used to have, into their data [9]. Since the data is also getting more detailed and diverse, (*ii*) it becomes harder for them to make exact definitions of interesting knowledge (i.e. what they are looking for) [14]. Apparently, these two *deficiencies* form the need for data exploration within scientific data analysis. The *explorer* (i.e. domain scientist) step by step explores the data, until he is satisfied with his understanding of data or he finds out some interesting knowledge. At the same time, the data management system involved should respond to his tasks near- instantly in order to provide interactive exploration.

The predominant way to organize scientific data is standard formatted files in a file repository. In addition to the actual data, these files contain self-descriptive measurements, called *metadata* (e.g. parameters, properties, and ways of acquisition, etc.) [4, 9]. We use the phrase *actual data* for the data other than metadata. Hence, (big) actual data (e.g. time-series, images, sequences, etc.) is accompanied by (small) metadata describing it. During processing, metadata can be accessed in order to identify actual data to be analyzed [20, 7, 16]. Hence metadata shines as a significant common aspect of scientific datasets, as Jim Gray et al. also concluded in [9].

These scientific datasets form a major part of the big data, and scientists are not data management specialists. These reasons have caused scientific data to attract some attention from database research [20, 9, 4, 14, 22, 11, 21, 8, 7, 16]. However, this research mainly focuses on how to integrate scientific data into databases, identifying the requirements, or giving array-support. None of them target solutions specifically for efficient interactive data exploration.

Interactive data exploration. If a scientist had full insight into the data and exactly knew what he is looking for, he would just run one query to extract or compute the interesting knowledge. However, scientific data analysis is not a one-query task. It involves exploration of the data space by a lengthy sequence of queries. Each of these queries reveals relatively small data areas (i.e. *data of interest*), as compared to the entire data space (See Figure 1a).

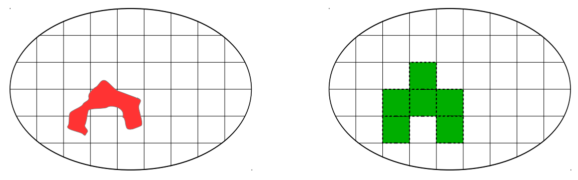
(a) Example data of interest (b) Corresponding files of interest

Figure 1: An illustration of the data space formed by a file repository. The ellipse represents the entire data space and each tile represents data in each distinct file. For simplicity of presentation, the data in the files are non-overlapping.

Each query is determined by how the explorer interprets the insight and experience gained throughout the execution of previous subsequence of queries. This requires each query's *informativeness* to be worth the time and resources spent executing it in order to shorten the sequence of queries or shorten running time of a next query. Moreover, the first query is typically *a quick look* into potential data of interest. Then the explorer might decide to zoom in (or out), or he simply moves to other potential data of interest. The straightforward solution to the need of efficient exploration is using a normal relational database to answer a sequence of queries, because databases have been capable of that for so many years.

Problems. Although a database system will be able to handle explorative queries, it still has the following problems towards efficient interactive data exploration:

(i) *Lack of metadata exploitation.* Without any insight into input data files, all available files have to be considered "relevant" for a given query. Metadata provides a medium to understand (i.e. gain insight into) the data, thus central to scientific data access. It should even be extended [9].

(ii) *Long initialization.* A quick look is not possible [10]. Gaining the first insight with the first query answer takes a long time (i.e. *data-to-insight* time). This is because current databases require data to be *ingested* (or *loaded*) into their internal structures before querying, to provide efficient query processing. However, loading all files in the repository eagerly up-front is an overkill, when a query can be answered with only the files that contain its data of interest (i.e. *files of interest*, see Figure 1b).

(iii) *Long exploration.* Due to the two deficiencies, the explorer simply can not know a priori how to phrase the query in such a way that it is fast to process with limited resources, still produces informative answers, and does not unnecessarily re-explore data of interest that has already been explored. The long running queries and long subsequence of queries with (unwanted) large (or empty) result sets, are not informative enough and/or the time spent on them does not pay off [14].

Approach. To address the above problems, we propose a novel two-stage query execution paradigm. We break the query execution into two stages, while still using a single query plan. In the first stage we execute the part of the query plan that uses the metadata. Whereas, in the second stage we execute the part of the query plan that uses the actual data (i.e. the rest of the query plan).

We load only metadata up-front. Files of interest are ingested in the second stage of execution, wherever and whenever we need them (i.e. lazily), without the explorer noticing (i.e. automated). We call this *automated lazy ingestion* (ALi). Additionally, within the first stage, we gain insight about explorer's interest and the query's informativeness. According to this, we perform a run-time query optimization between the two stages.

The two-stage query execution (including ALi) promises new research directions towards efficient interactive scientific data exploration, as it creates breakpoints within the queries. These can be exploited (further than ALi) to increase the efficiency and interactivity. Our new paradigm is embedded in the database kernel and is part of query processing and storage engine. Furthermore, it does not require any change in the querying front-end. In this paper, we sketch the research space towards realizing such database kernels.

2. RELATED WORK

The landscape of software architectures for scientific applications consists of large collections of programming scripts (i.e. legacy tools), workflow systems, and middleware solutions. Legacy tools mostly apply file-at-a-time procedural data analysis. So, they do not scale to the emerging dataset sizes, and they are specialized for specific tasks. Scientific workflow systems (like Kepler [6] and Taverna [3]), generalize these tools. They provide a visual programming front-end for construction of complete analysis pipelines. Designing a complete analysis pipeline and predefining the data of interest is something that a scientist is able to do only after enough exploration, though. Middleware solutions (e.g., [20, 7, 16]) use databases only for metadata querying. Then the middleware application finds and opens the resulting files for further analysis. Identification of actual data of interest is provided, whereas further in-database processing of this actual data is not. Thus, such solutions prevent continuous exploration by having separate analysis steps for the files of interest. There is also work to dedicate databases to scientific data. SciDB [21] and SciQL [22] make arrays first class citizens of databases. They do, however, neither exploit any metadata nor provide solutions for long exploration. The problem of long up-front data ingestion from a file containing tabular data, is addressed by the NoDB system [5]. Though seems similar, our ALi addresses a fundamentally different ingestion problem. We ingest files needed per query on the level of a file repository, whereas they ingest rows needed per query on the level of a file. Hence the approaches are orthogonal and even complementary.

3. APPROACH

Our goal is to make proper use of the given metadata, reduce the time spent for the entire scientific data exploration process, including the initialization time. For that, we load only metadata eagerly, and we break the query execution into two stages. Since we do not change the querying front-end, we still use a single query plan for a single query. Consequently, the first stage runs only the partial query plan that operates on metadata directly or indirectly. Therewith, the set of files of interest is computed. Once that is known to the system, files of interest are ingested (i.e. ALi) within the second stage, unless they have already been ingested and are in the cache. This satisfies the need for actual data to answer the query. Thus, using metadata, we limit the number of files needed to be accessed. The remainder of the section gives a first sketch of concepts and design elements with challenges towards realizing the approach.

Schema. Every relational database requires a schema before any other operation. A scientific relational database schema contains a set of relations/tables T. It consists of a set of metadata tables M (i.e. database tables that keep metadata) and actual data tables A (i.e. database tables that keep actual data). Thus, $T = M \cup A$.

Access Paths. *Access paths* represent ways to retrieve tuples from a table. In a relational database, an access path is either a scan or an index-scan. We enrich this set by adding three more access paths, namely *result-scan*, *cache-scan* and *mount*. The result-scan operator accesses the result set of a query (sub-)plan. The cache-scan operator accesses the data that was ingested from an external file and kept in the cache. The mount operator is responsible for ALi. It extracts, transforms (to comply with database schema) and ingests actual data from individual external files. We prefer the name 'mount' for this operation rather than 'load' because we do not actually load the actual data into the actual data tables. Instead, we make them accessible to the system (i.e. mount them) as dangling partial tables and unmount them after the query, unless we decide to cache them.

Relational Query Plan. When a query, written in a language such as SQL, arrives at the database, it is translated into a relational query plan (i.e. tree). Such a plan is then optimized using a set of rewrite rules (e.g. combine selections and cross-products into joins, push down selections, etc.). The two-stage query execution paradigm does not change these. Hence, the queries are the same as in the case where the database is eagerly loaded with all data up-front, and the same initial relational query plan is produced for the same query. Then, we *logically* decompose the relational query plan into two,

$$Q = Q_f \triangle Q_s$$

such that Q_f is always evaluated before Q_s, where Q_f is the highest branch in the relational algebra tree that has only metadata table scans as the leaves (i.e. metadata branch), and Q_s is the rest of the query plan Q.

In general, the usual query optimizers might end up with any kind of join order according to the order of joins between each metadata table m and each actual data table a. However, since we need to guarantee in our approach that we process metadata first (i.e. Q_f), we apply additional plan rewrite rules such as,

$$m_1 \bowtie (a_1 \bowtie m_2) \rightarrow a_1 \bowtie (m_1 \bowtie m_2)$$

considering a schema where $M = \{m_1, m_2\}$ and $A = \{a_1\}$.

These rules aim to form Q_f and Q_s. They collect joins on metadata tables together and push them down in the relational query plan. They use the associativity and commutativity properties of the join operation. In general, our algorithm detects the join order, and then rewrites any join order into the pattern

$$a_1 \bowtie (a_2 \bowtie (... \bowtie (a_y \bowtie (m_1 \bowtie (m_2 \bowtie (... \bowtie (m_{x-1} \bowtie m_x)...))))...))$$
$$\text{where } \; M = \{m_1, m_2, ..., m_x\} \; \textit{and} \; A = \{a_1, a_2, ..., a_y\}.$$

Q_s and Q_f might also share a smaller sub-plan. In that case, the sub-plan is not replicated. Instead, we enable Q_s to access the result of the sub-plan. Additionally, Q_f might contain cartesian products instead of joins, depending on the schema design. Furthermore, it is not needed to form Q_f and Q_s, unless the query refers to both metadata and actual data.

After logical plan optimizations, we also do optimizations on the physical query plan. To provide the ALi functionality, for each actual data table a in A, we apply additional rewrite rules such as,

$$scan(a) \rightarrow \cup_{f \in \textit{result-scan}(Qf)} \begin{cases} \textit{cache-scan}(f), & \text{if } f \in C, \\ \textit{mount}(f), & \text{otherwise} \end{cases} \quad (1)$$

where C is the set of files, data of which is in the cache, and each f is a file of interest. Data of the mounted files might be cached depending on the cache policy. If found beneficial, further query optimizations can be conducted by using rewrite rules such as pushing down selections or groupings into unions, e.g.

$$\sigma_p(scan(a)) \rightarrow \cup_{f \in \textit{result-scan}(Qf)} \begin{cases} \sigma_p(\textit{cache-scan}(f)), & \text{if } f \in C, \\ \sigma_p(\textit{mount}(f)), & \text{otherwise.} \end{cases}$$

Moreover, selections can be combined with mounts and/or cache-scans, creating two more access paths. They can then be employed if they are found beneficial by the usual query optimizers. Combined selections with cache-scans even lets the cache storage be tuple-granular rather than file-granular. This leads to a more precise cache management. Though, we need to mount the whole file even if there is one required tuple missing in the cache. Thus, it leaves a question behind, when and how one cache granularity is better than the other for explorative scientific workloads. Furthermore, since these rewrite rules require the result of Q_f to be computed, we apply them between the first and the second stage of execution, leading to a run-time query optimization phase.

Physical Query Execution. The newly designed query execution process consists of four physical steps. These are a compile-time query optimization phase, partial execution of the query (i.e. the first stage), a run-time query optimization phase, and lastly, execution of the rest of the query (i.e. the second stage).

```
SELECT AVG(D.sample_value)
FROM F JOIN R ON F.uri = R.uri
  JOIN D ON R.uri = D.uri AND R.record_id = D.record_id
WHERE F.station = 'ISK' AND F.channel = 'BHE'
  AND R.start_time > '2010-01-12T00:00:00.000'
  AND R.start_time < '2010-01-12T23:59:59.999'
  AND D.sample_time > '2010-01-12T22:15:00.000'
  AND D.sample_time < '2010-01-12T22:15:02.000';
```

Figure 2: Query 1

We visit each of these steps, using an example query running through the steps. Query 1 (see Figure 2) expresses the short term averaging task performed by seismologists while hunting for interesting seismic events in a seismic file repository. Each seismic file contains multiple records. A record represents the sensor readings over a consecutive time interval, i.e., a time series. Think of a simple underlying schema that consists of three tables. One metadata table F is for file-level metadata, and another R for record-level metadata, whereas a single actual data table D stores time series data points (i.e. tuples of sample time and sample value) from all files and records. Only metadata tables (i.e. F and R) are loaded eagerly in our approach, actual data tables (i.e. only D here) are empty. Detailed information about the dataset is in section 4.

In the *compile-time query optimization* phase, usual compile-time optimizations (e.g. pushing down selections and projections, etc.) are performed normally. We additionally reorganize the relational query plan to form Q_f. At the end of the phase, the relational query plan for Query 1 looks like

$$\gamma_{AVG(D.sample_value)}(\sigma_{p_3}(scan(D)) \bowtie \boldsymbol{(\sigma_{p_1}(scan(F)) \bowtie \sigma_{p_2}(scan(R)))})$$

where p_1, p_2, and p_3 represent the conjunction of selection predicates on tables F, R, and D respectively, and projections are ignored, for readability reasons. Finally, we mark the tree branch Q_f (here depicted in bold face), so that we know where to break the execution. The non-bold plan represents Q_s.

In *the first stage of query execution*, only Q_f is executed. At the end of this stage, the files of interest are identified, and collected as a list of file URIs. Say, there are three of them for Query 1, denoted by f_1, f_2, and f_3, and f_3 is in the cache. Although not the case in Query 1, if a query browses only metadata (i.e. does not refer to actual data), then the first stage of execution is naturally enough and the query is answered without any actual data ingestion.

In *run-time query optimization* phase we can make use of the insight gained in the first stage of the execution. To provide ALi, rewrite rule 1 is applied as default. The resulting plan looks like

$$\gamma_{AVG(D.sample_value)}(\sigma_{p_3}(\textit{mount}(f_1) \cup \textit{mount}(f_2) \cup \textit{cache-scan}(f_3)) \bowtie \boldsymbol{Q_f})$$
$$\text{where } \; \textit{result-scan}(Qf) = \{f_1, f_2, f_3\}$$

The part of the plan that has already been executed, is depicted in bold face. The non-bold plan now represents the rewritten Q_s.

Further optimization space can be explored here. To illustrate, an obvious strategical decision is about whether (a) we should merge the actual data taken from each file (if there is more than one file of interest) into comprehensive table(s) and then apply the higher operators in the plan in bulk fashion or (b) we should run higher operators on sub-tables and then merge the results.

In *the second stage of query execution* the paused execution continues with Q_s and ALi is physically performed. Then the remaining operators are executed. The query result is returned as usual.

This part of the work which provides ALi can also be considered as further realization of the concept of just-in-time access to data of interest, envisioned in [11] and [12]. Moreover, the approach applies the paradigm mainly within the query plan rewrite and execution layer of the DBMS architecture. Characterizing the trade-offs of applying the paradigm in different places in the architecture is another research direction and beyond the scope of this paper.

4. PRELIMINARY EVALUATION

To evaluate the potential of our approach, we have chosen seismology as the scientific domain. In seismology, SEED [1] is the most widely used standard file format to transfer waveform data among seismograph networks.

Scientific Data. A SEED volume mostly consists of waveform time series. These are highly compressed – see Table 1. Moreover, there are control headers keeping the metadata. We use the Mini-SEED (mSEED) variant. It reduces the SEED metadata to the most widely used subset. The common size range of an mSEED file is from 4 KB to several MBs. The normalized schema of the database is made of three tables; *F*, *R*, and *D* as explained in section 3.

Queries. Explorative analysis queries over the seismic data varies from simple data retrieval of an entire record for visualization, to averaging and outliers detection. Our sample queries here consists of the two queries taken from [13]. Query 1 (see Figure 2), computes the short term average over the data generated in Istanbul (`ISK`) via a specific channel (`BHE`). Query 2 has the same `FROM` clause as Query 1, but retrieves a piece of waveform (i.e. `D.sample_time, D.sample_value`) from all channels at a given station (i.e. a selection predicate on `F.station`), to visualize the data around a potentially interesting point (i.e. selection predicates on `D.sample_time`).

Experimental Setup. Our experimentation platform consists of a desktop computer equipped with a 3.4 GHz quad-core Intel Core i7-2600 CPU (hyper-threading enabled), 8 MB on- die L3 cache, 16 GB RAM, and a 1 TB 7200 rpm hard disk. The machine runs a 64-bit Fedora 16 operating system (Linux kernel 3.3.2). We employ MonetDB [2] as relational database. It is an analytical column-store with support of a scientific declarative query language: SciQL [22]. We extended MonetDB with a base implementation of two-stage query execution including ALi. The times reported are average execution times of three identical runs.

We realized two different ingestion approaches for comparison; *Ei* (eager ingestion) and *ALi*. In Ei, we extend MonetDB with the required functionality to understand mSEED files, extract, and load their data into the database tables inside the DBMS server. The entire input repository is loaded eagerly up-front using the functionality. In ALi, we use our new query execution paradigm plus we load only the metadata. As explained, the actual data will only be mounted if needed during the second stage of the query execution. The extraction of (meta)data from mSEED files is realized with the libmseed library [15].

When the data is eagerly loaded up-front, indexes are definitely needed for query performance, even though they increase the data loading costs and storage requirements. So, Ei creates primary and foreign key indexes before querying starts. Foreign key relations between the tables can be observed in the `FROM` clause of Query 1. For ALi, we do not build any indexes.

For now, the data ingested by ALi is discarded as soon as the query has been evaluated. While caching ingested data might avoid repeated mounting of the same files, the chosen approach inherently ensures up-to-date data. These require a detailed study and are not addressed in this preliminary design.

For the preliminary evaluation, we create a dataset by randomly selecting 5000 files of 161329 files from the year 2010. Table 1 lists some characteristics of the input repository used. This collection of mSEED files is copied from the ORFEUS file repository [17], which currently has over 3,5 million files. The copied sub-repository is stored on a local server and is accessible via NFS.

records per table			size			
F	*R*	*D*	mSEED	MonetDB	+keys	ALi
5,000	175,765	660,259,608	1.3 GB	13 GB	9 GB	10 MB

Table 1: Dataset and sizes

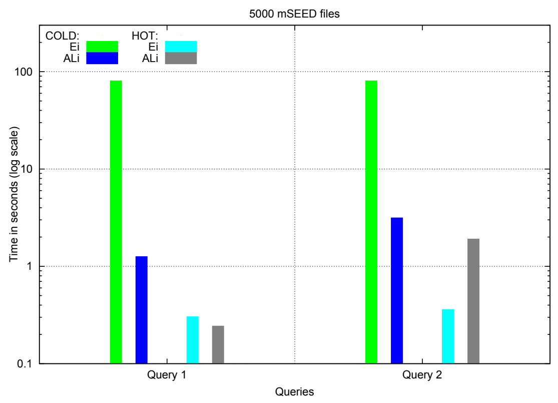

Figure 3: Querying 5000 files

Results. Table 1 lists the size of the original mSEED files, the size after loading into MonetDB without indexes, the additional storage required for the primary and foreign key indexes (together representing required space for Ei), and the size of the loaded metadata in the ALi case. Due to decompression and explicit materialization of timestamps, the MonetDB storage is much larger than the mSEED repository. The time for loading in Ei and in ALi case is directly proportional to the sizes that have to be ingested eagerly. Plus, actual data is also decompressed in Ei. In the experiments we observe that building the primary and foreign key indexes take four times longer than actual loading. They also do not pay off with a short sequence of queries. In addition, ALi provides more space-efficiency. The total amount of data stored in the data sources (i.e. repository) and the database is significantly less than that of Ei.

Figure 3 shows querying times for "cold" (i.e. right after restarting the server with all buffers flushed) and "hot" (i.e. with all buffers pre-loaded by running the same query multiple times one after another) runs. For cold runs, ALi definitely outperforms Ei for both queries. That is because the foreign key indexes in Ei have to be brought into main memory to compute the joins. With hot runs, ALi and Ei are in the same ballpark. For Query 1, ALi performed slightly better than Ei, because the time spent to mount files of interest is apparently shorter than the time spent to join entire actual data. Whereas for Query 2, ALi falls behind because Query 2 asks data from all channels of a location, making data of interest a lot larger than that of Query 1. These demonstrate that query performance of ALi is dependent on the size of data of interest. Intuitively, the best case is that the first stage of execution yields an empty set of files of interest, where no actual data is ever ingested. The worst case is that the data of interest is the entire repository, where then the performance becomes similar to the loading of Ei.

While the primary purpose of these experiments are to show applicability, the results obtained are encouraging, considering that in science large amount of data is ingested on a daily basis. This gives us the motivation to consider the non-addressed challenges.

5. CHALLENGES

The two-stage query execution enables new research directions towards efficient interactive scientific data exploration, as it creates breakpoints within queries in addition to the ones inbetween queries. Some obvious research questions it brings to the mind can be exemplified as follows. What is the most suitable query plan and breakpoint to minimize the data of interest? What is the best cache management for the data ingested lazily in a scientific data exploration setting? What is the best query execution strategy according to the query informativeness anticipated in the first stage of execution? How and when to extend the metadata (exploitation) as Jim Gray advised? When and how do we re-implement (some) strict requirements of today's databases while answering the above questions? Below, we sketch several research directions created by some of the above research questions.

Interactive query execution. A long running query affects interactivity negatively. For example, remember the worst case of ALi, explained in section 4. The database might return a completely incomprehensible answer of millions of rows with arbitrary numbers. The response time might be even days depending on how poor the query is phrased. Therefore, the explorer has to phrase a worthwhile informative query. On the other hand, to phrase such a query, the explorer needs to be insightful about the data and what he is looking for, which is not the case during exploration. After he fires it to the database, why can't he have a way to interfere with his own query's destiny (i.e. execution), when he sees that his query is running longer than he expected? Since we have a two-stage query execution paradigm and we gain some knowledge in the first stage, we can also anticipate the query's informativeness. This makes it necessary to quantify the query informativeness. After modeling and evaluating that, we can let the explorer learn expected time and resource consumption of his query at the breakpoint and let him even change the destiny of his query, interacting with the system. This addresses the problem of long exploration and might provide the fundamentals to realize the visions of one-minute database kernels [14] and queries as answers [18]. This line of research goes towards breaking some strict requirements of today's databases, namely, *correctness* and *completeness*.

Extending metadata. There are two types of metadata in general: given and *derived* metadata [19, 9, 4]. In general, derived metadata can be anything ranging from summary data (e.g. sum, average, max, etc.) to analyzed data (e.g. gaps, overlaps, etc.) [19]. Explorative analysis queries might refer to and require both of them. Running separate queries on the data to generate such derived metadata is unnecessary if that data has already been explored. Plus, having some derived metadata already computed and stored in the database before such a query comes will increase the query performance. It may even eliminate some of the long running queries. Thus, we can derive metadata as a side-effect of ALi or actual data processing, without the explorer noticing, in order to address lack of metadata exploitation and long exploration.

Generalization. Some scientific data might not be structurally expressible in relational tables (e.g. hierarchical data). Even if we manage to express them in tables (and arrays) within a database, their ingestion is format-specific. Although mapping of data to tables is done only once for a file format, different scientific domains usually have different formats. Usually scientists or scientific software do, however, similar operations in different fields of science to reach and manage their own scientific data, as explained in section 1. First, we can integrate all the common operations into the database system. Second, we can design a generalized medium for the scientific developer. Therewith, he can define domain- and format-specific mappings and extractions in a simpler way instead of someone writing code for the database kernel for every other scientific format. These two are open challenges for generalization.

Multi-stage query execution. Ideally, we can even go for a 'multi-stage query execution' paradigm where the system tries to anticipate the query informativeness in more than one place during query execution. It even tries to ingest in more than one place during execution. Consequently, we can allow more interactivity, which goes towards the user having full control over his query's destiny, even after the query leaves him and comes to the database.

6. CONCLUSIONS

Efficient interactive data exploration need is a rich source of open research challenges. Here, we identified current obstacles towards it; lack of metadata exploitation, long initialization and exploration time. To these ends, we aim to design a database kernel that breaks the query execution into two stages. The first stage is dedicated to metadata processing. So that the metadata is treated separately and can be exploited easily. In the second stage, ingestion and processing of actual data are performed. Hence the data ingestion is integrated into the query execution, leading to query-driven on- demand loading. Preliminary results showed that the initialization time is reduced by orders of magnitude, while not losing from the query time. There is more work to be done, though. For example, the compile-time and run-time optimization space has to be fully explored and evaluated. Moreover, non-informative long running explorative queries need to be targeted using the breakpoint between two stages of execution.

7. REFERENCES

[1] *Standard for the Exchange of Earthquake Data*. Incorporated Research Institutions for Seismology, February 1988.

[2] MonetDB, Column-store Pioneers. www.monetdb.org, 2013.

[3] Taverna workflow management system. www.taverna.org.uk, 2013.

[4] A. Ailamaki et al. Managing scientific data. *Commun. ACM*, 53(6):68–78, June 2010.

[5] I. Alagiannis et al. NoDB: Efficient Query Execution on Raw Data Files. In *SIGMOD*, 2012.

[6] I. Altintas et al. Kepler: An extensible system for design and execution of scientific workflows. In *SSDBM 2004*.

[7] P. Baumann et al. The multidimensional database system RasDaMan. *SIGMOD Rec.*, 27(2):575–577, 1998.

[8] J. B. Buck et al. Scihadoop: Array-based query processing in hadoop. In *SC 2011*, page 66. ACM, 2011.

[9] J. Gray et al. Scientific Data Management in the Coming Decade. *SIGMOD Record*, 34(4), 2005.

[10] S. Idreos et al. Here are my data files. here are my queries. where are my results? In *CIDR 2011*.

[11] M. Ivanova et al. Data vaults: A symbiosis between database technology and scientific file repositories. In *SSDBM 2012*.

[12] M. Ivanova et al. Data vaults: Database technology for scientific file repositories. *CiSE*, PP(99):1–1.

[13] Y. Kargin et al. Instant-On Scientific Data Warehouses — Lazy ETL for Data-Intensive Research. In *BIRTE*, 2012.

[14] M. L. Kersten et al. The researcher's guide to the data deluge: Querying a scientific database in just a few seconds. *PVLDB Challenges and Visions*, 2011.

[15] The Mini-SEED Software Library, libmseed. http://www.iris.edu/software/libraries/, 2013.

[16] V. Megler et al. Finding haystacks with needles: ranked search for data using geospatial and temporal characteristics. In *SSDBM 2011*.

[17] ORFEUS. Seismology Event Data (1988 - now). ftp://www.orfeus-eu.org/pub/data/pond/, 2013.

[18] T. Sellam et al. Meet charles, big data query advisor. *CIDR 2013*.

[19] A. Shoshani et al. Characteristics of scientific databases. In *VLDB*, pages 147–160. Morgan Kaufmann Publishers Inc., 1984.

[20] E. Stolte et al. Scientific data repositories: Designing for a moving target. In *SIGMOD 2003*.

[21] M. Stonebraker et al. Requirements for Science Data Bases and SciDB. In *CIDR*, 2009.

[22] Y. Zhang et al. SciQL: bridging the gap between science and relational DBMS. IDEAS '11, pages 124–133. ACM.