The DBMS – your Big Data Sommelier[⋆∗]

Yağız Kargın Martin Kersten Stefan Manegold Holger Pirk
Database Architectures Group, Centrum Wiskunde & Informatica (CWI), Amsterdam, The Netherlands
<*first*>.<*last*>@cwi.nl

***Abstract*—When addressing the problem of "big" data volume, preparation costs are one of the key challenges: the high costs for loading, aggregating and indexing data leads to a long *data-to-insight time*. In addition to being a nuisance to the end-user, this latency prevents real-time analytics on "big" data. Fortunately, data often comes in *semantic chunks* such as files that contain data items that share some characteristics such as acquisition time or location. A data management system that exploits this trait can significantly lower the data preparation costs and the associated *data-to-insight time* by only investing in the preparation of the relevant chunks. In this paper, we develop such a system as an extension of an existing relational DBMS (MonetDB). To this end, we develop a query processing paradigm and data storage model that are *partial-loading aware*. The result is a system that can make a 1.2 TB dataset (consisting of 4000 chunks) ready for querying in less than 3 minutes on a single server-class machine while maintaining good query processing performance.**

I. Introduction

While data is growing bigger and bigger, this growth is hardly due to manual creation of data items – machine-generated data is responsible for the lion's share of today's data volume [1]. Fortunately, automatically acquired data is much more regular than manually created data due to the inherent correlation of the data items stemming from the same source. This results in a high degree of spatial locality that can be exploited when managing such "big" data.

In practice, ("big") data often comes in *semantic chunks*: collections of hundreds or thousands of data items that share some common characteristics and are co-located when stored (e.g., in files). A typical example is the position of a camera in traffic or object monitoring: all data points collected by this sensor have the same geo co-ordinates, angle, crop, sensor type and configuration. This data, often called *metadata*, is perhaps the most important data because it gives meaning to the deluge of data [2].

Consequently, many analytical applications start by filtering their "big" data based on the (relatively small) metadata. In the light of this insight, it is reasonable to minimize the amount of preparatory effort spent on data that is likely to be filtered out based on its metadata. This is particularly important when, like in case of DBMSs, the mere presence of more data can hurt system performance through, e.g., reduced locality, growing index lookup times and less slack space for intermediate results. Naturally, only the system user knows which data is relevant to him. However, he expresses this knowledge in the queries he formulates. In combination with the metadata for each semantic chunk a system can exploit this knowledge to partially and adaptively ingest and prepare only the data that is relevant to the user.

⋆ Wiktionary: The person at an expensive restaurant who keeps the wine cellar and advises guests on a choice of wines.

∗ This publication was supported by the Dutch national program COMMIT. We thank Milena Ivanova for her helpful advices.

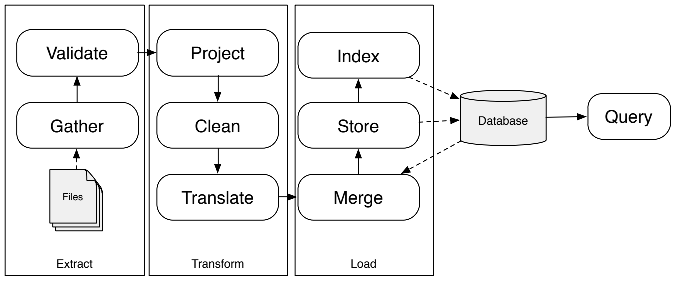


Fig. 1. A Common Data Preparation Pattern: Extract, Transform & Load (ETL)

This limits the storage space as well as the data preparation effort (Figure 1 illustrates the complexity of a typical data preparation pattern: ETL) to the semantic chunks that are of actual relevance to the user. However, since data has to be prepared at query evaluation time, this paradigm has an impact on all stages of the query evaluation process: plan generation, optimization, buffer management, catalog management and, last but not least, the query plan evaluation itself. While there has been some work on adaptive loading [3], indexing [4], view maintenance [5], [6], the chunked nature of "big" data has hitherto not been exploited.

In this work, we set out to develop a system that, like a good sommelier, stores the bottles (actual data) in the cellar (the file repository) but keeps the contents of the labels (the metadata) in his head to give advise and quickly retrieve bottles on demand. To this end, we make the following contributions:

- We develop a two-stage query processing model that distinguishes access paths to internal and external data when generating, optimizing and evaluating a query plan.
- We integrate the notion of partially loaded data with DBMS functionality like materialized views by treating such data structures as "partial" throughout the system.
- We provide a blueprint for the implementation of our approach in an existing, full-fledged DBMS: a description of our extensions of MonetDB to make it "partial-loading-aware". To show the benefits of this approach, we perform an extensive evaluation on one of the "biggest" datasets there is: scientific sensor data.

We structured this paper such that we give a quick overview of chunked "big" data in Section II. In Section III we illustrate how to exploit the obtained insights to develop an appropriate query processing model. In Section IV we explain our partial materialized view design. We describe the system's implementation in Section V. We evaluate this implementation in Section VI, review related work in Section VII, list future work in Section VIII, and conclude in Section IX.

II. Background

In this section, we discuss the characteristics of the data that is subject to our research, how we interpret the possible queries accordingly, and introduce an example dataset.

A. Characteristics of Data

Without any knowledge about the input data files, all available files have to be considered "relevant" for a given query. Fortunately almost all domains with chunked data have the notion of *metadata*, because there is simply a reason why the data is chunked, and metadata keeps that as information. Metadata is data that describes actual data in chunks and/or provides insight into the content thereof (e.g. parameters, properties, and ways of acquisition, summary data etc.) [7], [2]. We use the phrase *actual data* for the data other than metadata. The size of metadata is usually many orders of magnitude smaller than that of the actual data. Hence, big (actual) data (e.g. time-series, images, sequences, etc.) is accompanied by (small) metadata describing it. Although the definition of metadata might differ according to the domain, file format and use case, in most of the cases it is relatively straightforward to point out the metadata. For example, actual data items usually describe individual data points and come with, e.g., a timestamp and one (e.g. time-series) or many measured values (e.g. images). Whereas metadata items usually describe the actual data items (e.g. through which channel it is produced or mean value, etc.). Since metadata just describes actual data, there is a common usage of it. During processing, metadata can be accessed in order to identify actual data to be analyzed [8], [9], [10]. Hence metadata shines as a significant common aspect of scientific datasets, as Jim Gray et al. also concluded in [2].

The costs to acquire metadata does usually depend on the type of the metadata. There are two types of metadata in general: *given metadata (GMd)* and *derived metadata (DMd)* [11], [2], [7]. GMd is the describing data that is already accompanying the *actual data (AD)*. This metadata exists in the original dataset. It is mostly related to setup and configuration of the data generation, how the actual data is produced/generated, and/or in the case of scientific data, how the observations/experiments are measured etc. For example, location of the sensor, time interval of measurement, ways of acquisition, channel information, experimental parameters, configuration info, etc. DMd is again describing data, but that does not exist in the original dataset, and is not accompanying AD, usually because it requires further processing of AD and/or GMd. It could be, for example, summarizing AD (e.g. mean value), recognizing properties of AD (e.g. gaps/overlaps) or assessing its quality (e.g. data availability), etc.

GMd is ready to be used within the dataset. The cost to acquire such metadata is usually many orders of magnitude lower than that of AD. However, for DMd to be used in the analysis, it has to be derived by acquiring and processing AD (and/or GMd).

Type of Query	Query Refers to
T_1	GMd
T_2	DMd
-	AD
T_3	DMd & GMd
T_4	GMd & AD
-	DMd & AD
T_5	DMd & GMd & AD

TABLE I
Types of Queries

B. Interpretation of Queries

We look at the possible queries from a database point of view. We can categorize queries according to which types of data they refer to, to have a better understanding of what users would want from the database. Table I provides a list of types of queries and which types of data they refer to.

We do not focus on the "only AD" or "DMd & AD" type of query. For the latter, the DMd does not have to align with the chunks as GMd do. For example, a DMd item might be common in one small sub-chunk of every chunk. This happens if DMd items are computed regardless of the chunked nature of the data (i.e. computed without referring to GMd). Our approach does not benefit from such DMd for the queries directly referring to AD. So, our approach assumes DMd is computed from AD regarding the GMd. Hence for us, if the query is directly referring to AD, it has to refer to the GMd. For the former, it does not make much sense to query only AD directly without using any given meta-information, when GMd is able to help identify the actual data of interest. If not even a single metadata item is available for a chunked dataset, then it is the same case as having all data in one chunk. If there exists some metadata, we did not observe much of AD-only querying examples in real life scenarios, either. It might happen only when the user wants to process the entire set of AD regardless of whether metadata exists or not. In this case, there is no alternative to paying the price for loading / accessing all AD, anyway. Since such cases are not the focus of the work we present here, we assume in this paper that AD is always referred to with GMd.

C. Example

For easiness of understanding, we provide an example of data in chunks: scientific data. We use it as a running example as much as we can in the rest of the paper. Our example domain is seismology.

In seismology, *SEED* [12] is the most widely used standard file format to exchange waveform data among seismograph networks. A SEED volume mainly consists of the waveform time series, which are highly compressed. For example, a SEED repository might require orders of magnitude more than the original storage size when loaded into a database [13]

```
SELECT AVG(D.sample_value)
FROM dataview
WHERE F.station = 'ISK' AND F.channel = 'BHE'
  AND D.sample_time > '2010-01-12T22:15:00.000'
  AND D.sample_time < '2010-01-12T22:15:02.000';
```

Fig. 2. Query 1

(see also Table II in Section VI). Moreover, they use domain-specific compression algorithms [12]. Additionally, a SEED volume has several ASCII control headers. The control headers contain the metadata. Here, we use the *Mini-SEED (mSEED)* variant, which reduces the SEED metadata to the most widely used subset. The sizes of an mSEED file commonly vary from 4 KB to several MBs. *Millions* of them are stored in remote file repositories with direct FTP access. An mSEED repository is an example of chunked data, where each chunk is a file in the repository. While this is a common case also in other disciplines, there are other cases, like BAM files used in genome sequencing, where huge files are internally chunked.

Each mSEED file contains multiple mSEED records. An mSEED record represents the sensor readings over a consecutive time interval, i.e., a time series. The normalized data warehouse schema, as proposed in [13], includes three tables, that are straightforwardly derived from the mSEED format. Two tables *F* and *S* hold metadata per mSEED file and mSEED segment, respectively. Whereas table *D* stores all the actual data points (i.e. tuples of sample time and sample value from all files and segments). Each mSEED file is identified by its URI, and contains the metadata describing the sensor that collected the data (`network`, `station`, `location`, `channel`) as well as some technical data characteristics (`data quality`, `encoding`, `byte_order`). Each segment is identified by its (segment) identifier (unique per file), and holds metadata such as `start_time` of the segment, sampling rate (`frequency`), and number of data samples (`sample_count`). The identifiers form also the foreign key relations between the three tables. For easy querying, we define a (non-materialized) view `dataview` that joins all three tables, *F*, *S*, *D*, into a (de-normalized) "universal table".

Seismic data analysis contains tasks that help hunt for interesting seismic events. Such tasks include finding extreme values over *Short Term Averaging* (typically over an interval of 2 seconds) and *Long Term Averaging* (typically over an interval of 15 seconds), retrieving the data of an entire record for visual analysis, etc. Query 1 (Figure 2), computes the short term average over the data generated at Kandilli Observatory in Istanbul (`ISK`) via a specific channel (`BHE`).

Seismologists also compute some properties, patterns, or parameters through analysis to develop better understanding of the data. Later they often even (re)use them in order to filter out or not to re-touch some actual data. These are typical examples of what we call derived metadata. Seismologists typically derive, for a specific time period, quality control parameters, power spectral density values, and/or summary metadata, etc. A typical reuse example is that they refer to quality control parameters later in order to skip low-quality data or take into account only very high-quality data. This is actually materialized views in the database terms.

```
SELECT D.sample_time, D.sample_value
FROM windowdataview
WHERE F.station = 'FIAM'
  AND F.channel = 'HHZ'
  AND H.window_start_ts >= '2010-04-20T23:00:00.000'
  AND H.window_start_ts < '2010-04-21T02:00:00.000'
  AND H.window_max_val > 10000
  AND H.window_std_dev > 10
```

Fig. 3. Query 2

For this example, we use summary metadata that is derived based on hourly windows. Table *H* holds this derived metadata and serves as a materialized view. In a simple and typical setting this kind of derived metadata contains maximum value, minimum value, mean value, and standard deviation of the actual data points that are generated in each station, through each channel, and per each non-overlapping window of one hour. The attributes of *H* are `window_station`, `window_channel`, `window_start_ts` (ts: timestamp), `window_max_val`, `window_min_val`, `window_mean_val`, `window_std_dev`, respectively. Since *H* materializes summary metadata for each (station, channel, hour) triple, the first three attributes form the primary key of *H*. Again for easy querying, we define a (non-materialized) view `windowdataview` that joins all four tables, *F*, *S*, *D*, *H*, into another (de-normalized) "universal table". [1]

While hunting for interesting patterns, an example query for the use of this derived metadata occurs, whenever the scientist would like to retrieve waveform data where volatility is very high in high amplitude value levels. As a concrete example, Query 2 (Figure 3) brings the waveform data from station `FIAM` and channel `HHZ` of the hours in a given time interval, where maximum amplitude in that hour and standard deviation in that hour are greater than `10000` and `10`, respectively.

We use Query 1 and Query 2, which are T_4 and T_5 queries respectively, as our running examples through the rest of the paper. For more detailed information about the data, schema, and queries; please refer to [13].

III. Two-stage Query Execution

Recall that our starting point is a file repository of automatically acquired naturally chunked "big" data, accompanied with some metadata that identifies and describes the chunks.

Our goal is to enable users to instantly use the conveniences of data management systems for their interactive exploratory data analysis, without the need to first ingest all their data into a DBMS. Instead of "*eagerly*" loading all actual data before being able to run the first query, we propose to exploit the metadata (both, given and derived) during query processing to "*lazily*" load only the required actual data transparently just-in-time during query evaluation.

The size of metadata is much smaller than the actual data and can thus rather quickly be loaded eagerly into the database once a file repository becomes available. We will discuss in

[1]While the non-materialized views `dataview` and `windowdataview` relieve the users from having to express these joins in each of their queries, the DBMS has to calculate the respective joins when evaluating queries over these views.

Section IV, under which circumstances we can assume that also derived metadata is readily available in the database.

To be able to exploit the metadata, 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. In a nutshell, in the first stage, the part of the query that uses only metadata (given or derived) is evaluated in order to determine the chunks (i.e. files) of actual data that need to be loaded to answer the entire query. In the second stage, these chunks of actual data are loaded — unless they have already been ingested earlier and are still in the cache — and the remainder of the query is evaluated. For now, we assume that all derived metadata is also computed and available in the database like the given metadata. In Section IV, we will justify this assumption by introducing an approach to make the derived metadata needed by the query available in the database. The remainder of this section provides a sketch of the concepts and design elements to realize this approach in any relational database.

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 a set of actual data tables A (i.e. database tables that keep actual data). Thus, $T = M \cup A$.

Example. F and S are our given metadata tables, D is the only actual data table. In our approach, only F and S are loaded eagerly, while D remains initially empty. Actual data is loaded into D only partially as required during query execution.

Logical Query Plan. An SQL query is translated into a relational query plan Q (typically taking the shape of a tree or directed acyclic graph). The relational query plan is optimized using a set of rewrite rules. Since we need to process metadata before any actual data, we have some plan requirements. We add some extra rules that make query optimizers produce the kind of plans our paradigm requires. To that end, we *logically* decompose the relational query plan into two parts,

$$Q := Q_f \triangle Q_s$$

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 tables as its leaves (i.e. metadata branch), and Q_s is the rest of the query plan Q. Figure 4(b) exemplifies a decomposed plan where the red sub-plan is Q_f and the rest of the plan is Q_s. The reason here is to make use of the metadata as much as possible to filter out actual data. Subsequently, we work on the required actual data in Q_s. Thus we do not have to load the actual data we never require, unlike the eagerly loaded database. To achieve this, we need to have specific join orders.

Example. 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. After the query is turned into a plan, usual compile-time optimizations (e.g. pushing down selections and projections, etc.) are performed. Join order is also determined. For this example, we

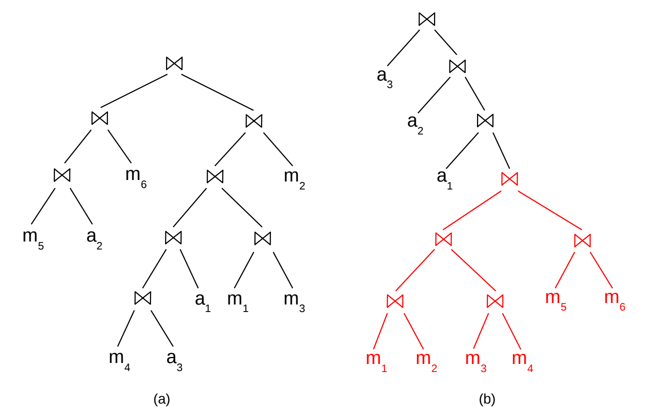


Fig. 4. An example initial join tree (a) and a possible final join tree (b)

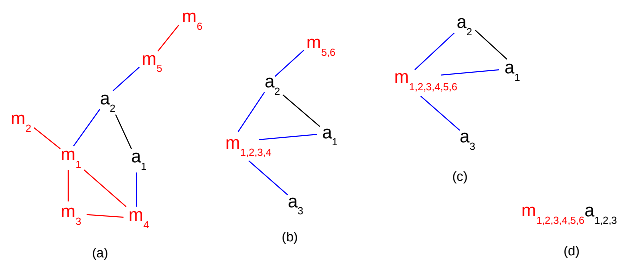


Fig. 5. Applying extended rule set on the given join graph (a). R1 transforms (a) to (b), R2 transforms (b) to (c), R3 & R4 transform (c) to (d).

assume a simple join order optimizer that takes only selections into account. Hence, it first joins F and D, the two tables with selection predicates. After optimization, the initial query plan might look like this:

$$\gamma_{AVG(D.sample\_value)}(S \bowtie (\sigma_{p_1}(F) \bowtie \sigma_{p_2}(D)))$$

where p_1 and p_2 represent the conjunction of selection predicates on tables F and D respectively, and projections are ignored, for readability reasons.

Join Order. In general, traditional query optimizers might end up with any kind of join order between each metadata table m and each actual data table a. However, since we need to guarantee that we process all metadata first, metadata tables should be pushed down into one sub-tree (i.e. Q_f) of the relational query plan tree. Since the rest of the plan tree (i.e. Q_s) should only be executed after Q_f completely evaluates, we do not allow bushy plan trees in Q_s. Figure 4(a) shows an example initial query plan. Whereas Figure 4(b) shows an example final plan we could end up with.

The arrangement of the join order is done by the join order optimizer. It takes an initial relational query plan, a set of rules to apply, a query graph [14] to know about join possibilities. As a result it produces a query plan tree which is optimized according to the given rules. Designing a join order algorithm from scratch is not the main point of this work and is also not necessary. We only extend the set of rules by classifying edges and vertices of the query graph. For that we distinguish between metadata tables and actual data tables (i.e. vertices) and join predicates connecting them (i.e. edges). For simplicity we use a coloring scheme. We color any vertex representing a metadata table red, and all others black. Then we color any edge between two red vertices red, any edge between two black vertices black, and any edge between a red vertex and a black vertex blue. Figure 5(a) shows a possible query graph for the initial query plan shown in Figure 4(a).

We add the following rules to the optimizer's set of rules:

R1) *Join on red edges first before anything else.*

R2) *Only if necessary, use cross-products to join all red vertices into one, before using any blue or black edges.*

R3) *Do not allow bushy plans containing black vertices.*

R4) *Join on black edges only if all other edges are used.*

Figure 5 illustrates how the given query graph would be transformed by applying the new rules. If a join order optimizer is run with above extensions, it might produce a join tree as in Figure 4(b). The red sub-tree of the plan could be in any join order (here depicted as bushy). The black sub-tree is in any linear join order (here depicted as right-deep). Our rule-set was motivated by the need to avoid the worst case, i.e., having to load all data for each query. The given set of rules is minimal in the sense that for each rule there is a query that requires this rule to avoid loading unnecessary data. For example, R2 prevents the access to a_2 without exploiting the metadata in m_5.

Example. We reorder the joins to form Q_f. After applying the set of rules, the logical query plan for Query 1 looks like

$$\gamma_{AVG(D.sample\_value)}(\sigma_{p_2}(D) \bowtie \boldsymbol{(S \bowtie \sigma_{p_1}(F))})$$

as all metadata tables are joined deeper than actual data tables in the tree. 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.

Physical Query Plan. After the logical plan optimization, we also do optimizations on the physical query plan. For that, we come up with new 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 *chunk-access*. 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 chunk-access operator is responsible for lazy loading of chunks. It extracts, transforms (to comply with the database schema) and ingests actual data from individual external chunks. We prefer the name "chunk-access" for this operation because any one-chunk (e.g. one-file) access strategy can be employed (e.g. full load or in-situ access). Moreover we might decide to cache the results of the chunk-access operator, to be accessed with the cache-scan operator later. We integrated a recycler [6] approach into our system as the caching mechanism for the lazily loaded chunks.

Example. To switch to the physical query plan, we place the access paths. For this example, we use the base access path, scan. Then the physical query plan looks like

$$\gamma_{AVG(D.sample\_value)}(\sigma_{p_2}(scan(D)) \bowtie \boldsymbol{(scan(S) \bowtie \sigma_{p_1}(scan(F)))})$$

where again the sub-plan in bold face depicts Q_f.

Run-time Query Optimization. To provide the "lazy" loading functionality, for each actual data table a in A, we also apply additional rewrite rules such as,

$$scan(a) \rightarrow \cup_{f \in \text{ result-scan}(Q_f)} \begin{cases} cache\text{-}scan(f), & \text{if } f \in C, \\ chunk\text{-}access(f), & \text{otherwise} \end{cases} \tag{1}$$

where C is the set of chunks that are cached in the database, and each f is a chunk of interest. Data of the lazily loaded chunks might be cached depending on the cache policy. By this rewrite rule, our approach minimizes the number of chunks to be loaded into the database. If found beneficial, further query optimizations can be conducted by using rewrite rules such as pushing down selections or groupings into unions, e.g.

$$\sigma_p(scan(a)) \rightarrow \cup_{f \in \text{ result-scan}(Q_f)} \begin{cases} \sigma_p(cache\text{-}scan(f)), \\ \qquad\qquad \text{if } f \in C, \\ \sigma_p(chunk\text{-}access(f)), \\ \qquad\qquad \text{otherwise.} \end{cases}$$

Moreover, if an in-situ access path is preferred (like NoDB [15]) as the one-file access path, selections can even be pushed down into the chunk-accesses and/or cache-scans. They could also be designed as another access path that the system has, in which case they can then be employed if they are found beneficial by the usual query optimizers. 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 this run-time query optimization phase.

Example. 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, of which f_3 is in the cache. During *run-time query optimization* we can make use of the insight we gained in the first stage of the query execution. To fully benefit from lazy loading, rewrite rule 1 is applied by default. The resulting plan looks like

$$\gamma_{AVG(D.sample\_value)}(\sigma_{p_2}(chunk\text{-}access(f_1) \cup chunk\text{-}access(f_2) \\ \cup\ cache\text{-}scan(f_3)) \bowtie \boldsymbol{Q_f}) \\ \text{where } result\text{-}scan(Q_f) = \{f_1, f_2, f_3\}.$$

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. In *the second stage* of query execution the paused execution continues with Q_s. First, non-cached required chunks are loaded. Then the remaining operators are executed. The query result is returned as usual.

Although not the case in Query 1, if a query browses only metadata (i.e. does not refer to actual data, so T_1, 2, and 3), then the first stage of execution is naturally enough and the query is answered without any actual data ingestion.

This part of the work which provides "lazy" loading can also be considered as further realization of the concept of just-in-time access to data of interest, envisioned in [16] and [17].

IV. Incremental metadata derivation

In the previous section we assumed that the derived metadata needed by the query is already available in the database. However, eager loading of derived metadata actually means eager computation of derived metadata (DMd) from given metadata (GMd) and actual data (AD). This is actually materializing a view over the entire data. However, a part of the actual data might not even be touched by the user. This is actually similar to the reason of why we load actual data lazily. Hence, similarly, we address this problem by computing DMd lazily on-the-fly and incrementally materializing it. That is, whichever DMd item we need, we compute them whenever we need and save it in the database. This is because we can further use it to filter out more chunks of actual data during query processing. This actually, translates into incrementally materialized views. There is already related work on partially materialized views: [18], [19]. In this part of our work, we integrate partially materialized views (by [18]) into our system, so that it makes use of lazy loading (i.e. partial-loading-aware), and it is exploited by the two-stage query model in order to filter out chunks. Since [18] (and also [19]) does not provide partial reuse capability, we sketch an algorithm to provide that later in this section.

When a DMd table is created, the user specifies the DMd attributes and describes how to compute them. We make use of the primary key attributes of the DMd table to keep track of which DMd items are already available in the DMd table (e.g. `window_station`, `window_channel`, `window_start_ts` values for our running example). We compute the required DMd before running the query. Then the query is able to run with two-stage query execution and benefit lazy loading. We explain how to do this in this section.

When a query comes into the system, we apply Algorithm 1 to answer the query and also realize on-the-fly metadata derivation, lazily. The algorithm consists of 7 steps. We explain it while running on the Query 2.

First of all, we find out the type of q, according to the types listed in Table I. If it is one of the types that does not refer to any DMd table, then q does not require any DMd. Thus, q is ready for execution (Step 7). If q refers to at least a DMd table, then we detect the predicates referring to the primary key attributes of the DMd table in Step 2. We do this by going through the predicates q has in the `WHERE` clause. If we look at Query 2, we notice that it is a T_5 query, because it uses `windowdataview` and also refers to the DMd table H. Hence, we identify the predicates in Step 2:

```
H.window_station = 'FIAM' AND
H.window_channel = 'HHZ' AND
H.window_start_ts >= '2010-04-20T23:00:00.000' AND
H.window_start_ts < '2010-04-21T02:00:00.000'
```

Note that the other two predicates on H are left out because they are not predicates on a primary key. Thus, the result of Step 2 defines a primary key space. In Step 3, we enumerate all possible values in that primary key space (i.e. **p**rimary key **s**et referred by the **q**uery q, PSq). PSq becomes the pointer to the DMd required by q. The PSq for Query 2 is:

Algorithm 1 On-the-fly Metadata Derivation

```
1:  current query: $q$
2:  (1) Find out the type of $q$.
3:  if it is not type 2, 3, or 5 then
4:      Jump to Step (7).
5:  (2) Find out the predicates referring to the primary key attributes
    of the DMd table in $q$.
6:  (3) Enumerate the PSq.
7:  (4) PSm is already materialized.
8:  if PSq is covered by PSm then
9:      Jump to Step (7).
10: (5) Find out the PSu.
11: PSu $\leftarrow$ PSq $-$ PSm.
12: (6) Compute the unavailable required DMd that is pointed by
    PSu and insert into DMd table via a T_2 query.
13: (7) Proceed with the execution of $q$.
```

window_station	window_channel	window_start_ts
FIAM	HHZ	2010-04-20T23:00:00.000
FIAM	HHZ	2010-04-21T00:00:00.000
FIAM	HHZ	2010-04-21T01:00:00.000

There is also the queried **p**rimary key **s**et that is already **m**aterialized in the DMd table (PSm). PSm is the pointer to the already available DMd predicates of Step 2 applied. For the example, let's assume one of the previous queries already required DMd of the day `2010-04-20`. Since DMd of that day is already in the DMd table, the PSm for the example is:

window_station	window_channel	window_start_ts
FIAM	HHZ	2010-04-20T23:00:00.000

In Step 4 we check if PSm covers PSq. If true, then q does not require any DMd to be computed. Thus, q is ready for execution (Step 7). However, if false, we need to find out what subset of PSq is not covered by PSm (Step 5). That subset of PSq is the **p**rimary key **s**et on which the **u**navailable required DMd is dependent (PSu). That is the pointer to the unavailable required DMd. Then the PSu for the example becomes:

window_station	window_channel	window_start_ts
FIAM	HHZ	2010-04-21T00:00:00.000
FIAM	HHZ	2010-04-21T01:00:00.000

In Step 6 we compute what PSu points to and insert that computed unavailable required DMd into the DMd table referred. This requires us run a T_2 query, which use the two stage query execution, and insert its results into the DMd table. This might require to employ lazy loading as well. After that, we proceed with the execution of q (Step 7), because the required DMd for q is in DMd table.

Since touching actual data might require lazy loading, this might make lazy loading dominate the cost of DMd computation. Kinds of DMd – which are not much costly to derive – where that is the case are derived as extra together with the required DMd. Thus, if we derive some metadata for a specific window, then we derive all possible metadata for that window. If the user do not want this, then he could separate the part of DMd which he does not want to derive, into other DMd tables. So, they could be only derived if those other DMd tables are queried.

V. System Realization

We realized our ideas in MonetDB [20]. MonetDB is an analytical memory-optimized DBMS based on columnar storage that maps logical relation algebra (SQL) to its internal physical column algebra *MAL* ("*MonetDB Assembly Language*"). In addition to a rule-based relational query optimizer, MonetDB also provides an optimizer infrastructure to rewrite MAL plans both statically during query translation. We enabled dynamic rewrite of MAL plans during query evaluation as well in the MAL interpreter. This can be envisioned as similar to self-modifying programs. Moreover, MonetDB has a layered modular and extensible software architecture that facilitates the extension of both the runtime functionality as well as the optimizer functionality by adding new modules.

To realize our ideas for a two-stage query execution including lazy loading, we extended MonetDB with three modules. The *Registrar* handles eager loading of given metadata, while the *Compile-time Optimizer* and *Run-time Optimizer* provide the two-stage query execution including lazy ingestion of actual data. While we introduced our paradigm generically on the relational algebra level in Section III, the "natural" place to realize it in MonetDB is it physical MAL algebra level and related extensible optimizer framework.

1) Registrar: When a new file repository is registered with the DBMS, the Registrar module iterates over all files in the repository, extract the given metadata and (bulk-)loads it into the respective tables (F & S in our seismology example). Exploiting MonetDB's multi-threaded architecture and multi-core support, multiple files can be handled in parallel.

2) Compile-time Optimizer: The Compile-time Optimizer takes care of splitting the query plan in two phases and creating the required join order by implementing the 4 additional rules introduced in Section III. Since MonetDB only considers linear join plans, rule R3 is given by default. The remaining rules could be realized by adjusting the scoring function of MonetDB's join-order optimizer. In addition, this optimizer also ingests a call to the Run-time Optimizer into the generated MAL query plan right between the two phases (Q_f & Q_s). The call receives the set of required actual data files as calculated by phase Q_f as argument (i.e. *result-scan*(Q_f)).

3) Run-time Optimizer: Evaluation of the plan as produced by the Compile-time Optimizer starts with phase Q_f that evaluates the query parts that access metadata and produces a set of required files. Once phase Q_f is finished, the plan call the Run-time Optimizer as described above. When called, the Run-time Optimizer modifies the remainder of the query plan as follows to load the required actual data. For each required file, it inserts a statement into the MAL plan to load its actual data into table D. In fact, we do not load all data into a single table, but rather into a separate (temporary) table per file. In this way, we can exploit MonetDB's existing multi-core support that (data-)parallelizes entire (sub-)plans rather than individual operators. In this case, rather than slicing a large base table, each file is a slice of table D.

Once done with all plan modifications, the Run-time Optimizer hands the control back to the MAL interpreter to finish the query evaluation, starting with executing the inserted statements to load the required actual data.

To achieve the caching of loaded actual data, we exploit MonetDB's existing intermediate caching component (the Recycler [6], i.e., no extra implementation is required.

Note that, while this approach nicely follows the bulk processing paradigm, it also inherits one of the drawbacks: the parallelization strategy is static. In our implementation, the degree of parallelization is determined by the number of semantic chunks that are selected in the loading phase. Therefore, a low number of selected chunks or skew among them can lead to underutilization of the machines CPU cores. This problem can be addressed by introducing the bulk processing version of a volcano-style exchange operator [21] to mediate the different degrees of parallelism in the loading and processing phases. However, this either requires a vector-wise exchange operator at runtime or a blocking, bulk-wise repartition operator. Since both of these come with different trade-offs, we consider a detailed, comparative study future work.

VI. Evaluation

To evaluate our approach that is designed for big data in chunks, we have chosen a scientific dataset, that contains (real) seismic sensor data.

A. Experimental Setup

Our experimentation platform consists of a server machine equipped with two 2.0 GHz 8-core Intel Xeon E5-2650 CPUs with hyper-threading enabled (i.e., 32 hardware threads in total), 20 MB L3 cache per CPU, 256 GB RAM, and 5.4 TB disk storage (3x SW RAID0). The machine runs a 64-bit Fedora 20 (Heisenbug) operating system (Linux kernel 3.12.10). We use MonetDB Feb-2012 version. All performance results are average of 3 runs.

sf	data of	records per table		
		`files`	`segments`	`data`
sf-1	40 days	160	2009	1,273,454,901
sf-3	4 months	484	7802	3,929,151,193
sf-9	1 year	1464	12566	11,912,163,036
sf-27	3 years	4384	74526	33,683,711,338

TABLE II
INGV Dataset

Datasets. We have taken our datasets from a repository of mSEED files which are collected from the Italian National Institute of Geophysics and Volcanology (INGV). We had access to a repository that contains 3-years of data from 4 stations (INGV has 15-years of data from 450 stations). We create 4 different datasets out of our INGV dataset with different scale factors in order to see the scalability of our approach. Table II shows characteristics of the datasets. They reside on HDD. The extraction of (meta)data from mSEED files is realized with the libmseed library [22]. That is, the *chunk-access* operator is realized through a full-scan of a chunk using the domain library.

Loading approaches. We compare the following 5 loading approaches. *Eager_plain* refers to just plain loading of mSEED files into the DBMS server directly. We extended MonetDB with the required functionality to read mSEED files directly. Eager_plain *eagerly* reads and loads the data from all given mSEED files before querying can begin. *Eager_index* refers to plain data loading plus indexing. In order to analyze the costs and benefits of creating and using primary and foreign key constraints, we consider constructing foreign key indices, which serve as join indices. Eager_index creates the respective indices after data loading. *Eager_dmd* refers to plain data loading plus indexing, and plus eager DMd computation (i.e. materializing a view). Since there are (actually 3 out of 5) query types referring to DMd, it could be beneficial to compute them as an initial investment. *Lazy* refers to our new 2-stage query execution approach that does lazy loading. Initially, the database server only extracts the metadata from all given mSEED files and loads it into tables `F` and `S`. Only during query evaluation in the beginning of the second stage of execution the actual time series data (only if not already cached) is extracted and loaded into a temporary `data` table and required records are taken. Then it is cached using the recycler cache. The granularity of the loading depends on the strategy chosen for loading of a single chunk. In this case the granularity is a file since we use the libmseed library to extract data from each file. Moreover, Lazy does not create any index on actual data. In addition, primary keys are all defined with all loading variants. Generally, we rely on the DBMS's standard mechanisms to verify and enforce constraints. With Lazy, we omit the foreign key constraints between the data table and the metadata tables, to avoid constraint verification whenever data is loaded (with each query). We consider this "safe", since all foreign key constraints in our framework are purely on system-generated keys, and thus enforced by design.

Queries. We use queries that are for the seismological tasks explained in Section II. We use 5 different queries, one for each query type listed in Table I. Our T_1 query joins GMd tables and has a selection predicate on `station`. Our T_2 query refers to the DMd table with selection predicates on `station`, `window_start_ts`. Our T_3 query is the same as the T_2 query except that it joins the DMd table with the GMd tables. Our T_4 query computes some aggregate function on the actual data values that are joined with GMd tables, with a selection on both GMd and AD. Finally, our T_5 query computes some aggregate function on the actual data values that are joined with GMd and DMd tables, with a selection on both GMd and DMd. We have 2 queries (T_4 and T_5) that directly refer to the actual data table (T_2 and T_3 queries only refer to AD indirectly through DMd). To see the effect of having a selection predicate on the actual data, we provide a selection on AD in the T_4 query, but not in the T_5 query. As a side note, direct AD-AD joins are also possible in our system using plain MonetDB query processing on top of our lazy-evaluation framework. However, it happened to be that they did not appear in our real life scenario.

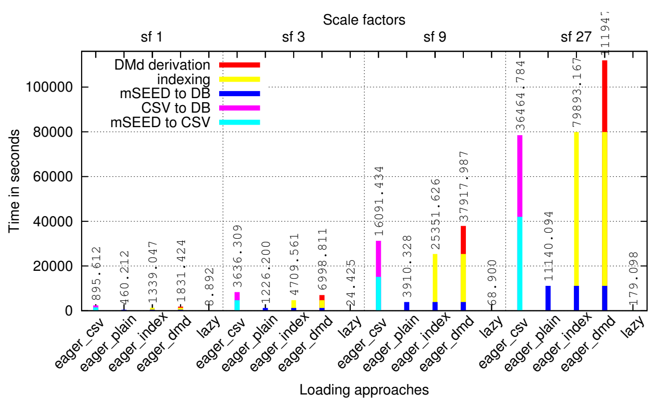


Fig. 6. Loading

B. Loading

Table III shows the size characteristics of the datasets: The size of the original mSEED files, the CSV files generated by Eager_csv (which writes mSEED data into CSV files and loads the CSV files with `COPY INTO` statement of MonetDB), the size after plain loading into MonetDB, the additional storage required for the primary and foreign key indexes, and the size of the loaded metadata only in the Lazy case. Due to decompression, explicit materialization of timestamps, the CSV files are much larger than the mSEED files. An interesting point is that our approach is more space-efficient than eager approaches. The amount of total data stored in the data sources and in the database is significantly less for the lazy case than that of the eager cases.

Figure 6 breaks down the initial investment costs into extracting data from mSEED files to CSV files, bulk loading data from CSV files into the DBMS, loading the data directly from mSEED files into the DBMS, creating primary and foreign key indexes, and computing and saving all DMd (i.e. as a materialized view).

The results confirm that although Lazy extracts metadata from all provided mSEED files, extracting only the metadata is orders of magnitude faster than extracting and loading all data. Also, Eager_plain is significantly faster than Eager_csv, mainly due to avoiding expensive serialization to and parsing from a textual (CSV) representation. Finally, creating primary and foreign key indexes more than doubles the preparation times for the Eager variants.

sf	size				
	mSEED	CSV	MonetDB	+keys	Lazy
sf-1	1.3 GB	45.5 GB	23.7 GB	18.9 GB	1.3 MB
sf-3	4.1 GB	139 GB	73.1 GB	58.5 GB	1.7 MB
sf-9	12.3 GB	429 GB	222 GB	176 GB	2.1 MB
sf-27	36.0 GB	1.2 TB	627 GB	502 GB	6.3 MB

TABLE III
INGV DATASET SIZES

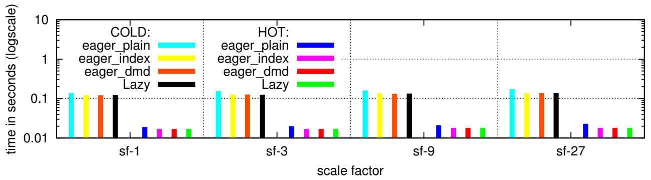
C. Domain Query Performance

We provide an upper and lower bound for single query (taken from domain experts) running times by providing results for both “cold” (right after restarting the server with all buffers flushed) and “hot” (with all buffers pre-loaded by running the same query multiple times after another) runs, respectively. For these experiments, we use the queries as they are typically used by seismologists. So, each type of query selects (the same) 2 days of data from (the same) one station, except T_1 query, which only selects GMd of one station and computes an aggregate. We experiment query performance of all types of queries with different scale factors on data loaded with different loading approaches. Figures 7a, 7b, 7c, 7d, 7e show the running time of each type of query, respective to their type number. T_1 query performance is in the same ballpark for all loading approaches and all scale factors, because there is just the (small) metadata being processed. For that reason plus the space limitation, we will omit results of the T_1 for next experiments. T_2 and T_3 queries running on Eager_dmd outperform Lazy by orders of magnitude. Query performance on Eager_dmd is in the order of milliseconds for those queries, which just returns values from the DMd view already materialized. This also helps Eager_dmd outperform querying on Eager_index for T_5 query for all scale factors. T_3 query give the same insight as T_2 query. It runs only a bit slower than the T_2 query in all scale factors, just because there is an additional join with GMd tables, which are rather small in size. For T_4 queries Eager_index and Eager_dmd perform similarly because the query does not refer to DMd. T_4 query performance on Lazy come to same ballpark with that of Eager_dmd, after being outperformed for T_2 and T_3 queries. For T_5 queries Lazy finally provide better query performance than Eager_dmd, because it filters out actual data in most of the files before the joins between metadata tables and actual data table.

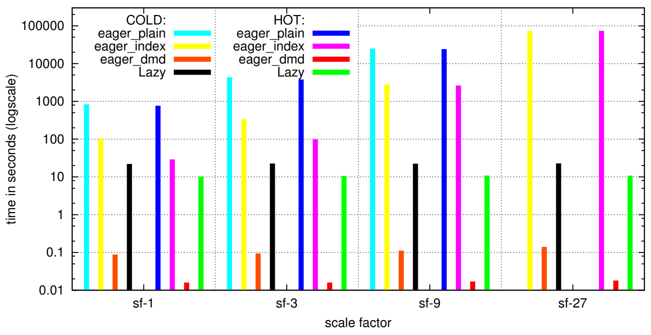
The benefit of exploiting foreign key indexes during query processing is visible. Eager_index always perform better than Eager_plain because all queries have joins. It pays off in a couple of queries, since constructing the join index is actually computing the join itself. However, the benefit is rather small for T_4 query, such that the investment pays off only after many queries, because T_4 query has a selection on actual data which limits the amount of data to be processed.

For sf-1 and sf-3, dataset plus the index completely fit in the main memory. For sf-9 either dataset or the index hardly fit in main memory. This makes Eager_plain and Eager_index querying roughly 10 times slower while scaling from sf-3 to sf-9 (though scale factor is 3 times greater) in running times of queries from T_2 to T_5 except T_4. This is again because T_4 query has a selection predicate on actual data. T_2 and T_3 queries even does not refer to actual data in the first place, although they have to process it to compute the DMd they refer to. For scaling from sf-9 to sf-27 it even gets worse for T_2, T_3, T_5 queries getting roughly 25 times slower. The reason is that neither the dataset nor the index fit in the main memory for sf-27, which degrades the benefit of the index significantly.

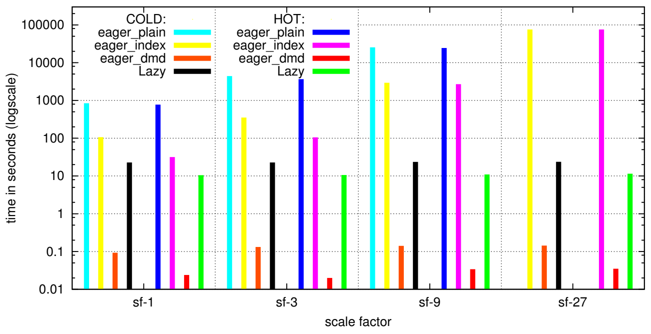
In contrast, lazy does not get affected by the scale factor. That is because each query is the same for all scale factors here. Thus, it has to load lazily the same amount of external



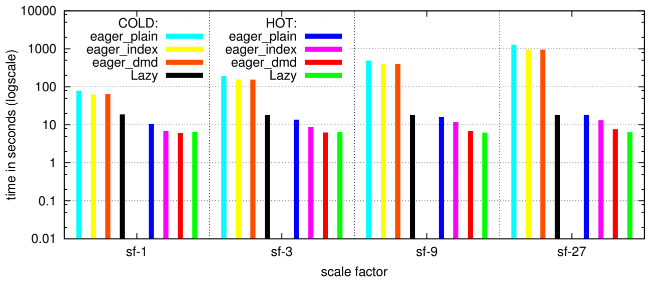
(a) T_1



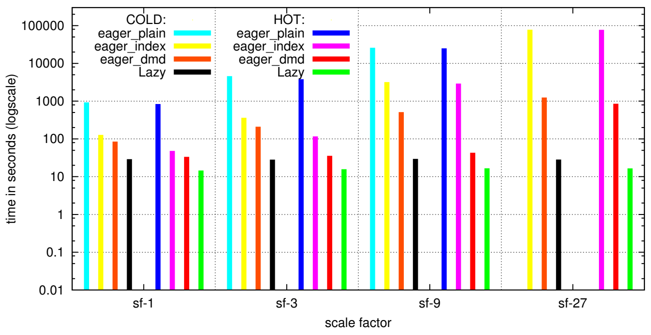
(b) T_2



(c) T_3



(d) T_4



(e) T_5

Fig. 7. Representative Upper and Lower Bounds for Single Query Performance

data. Since queries ask for 2 days of data from one station, Lazy has to load only 2 mSEED files lazily. Then it only has to process data of 2 mSEED files during the joins and afterwards. In conclusion together with the loading, up-front preparation time for the database is reduced by orders of magnitude, while query performance is not affected by this. What can affect performance of queries on Lazy is the selectivity of the queries, the effect of which we measure in the next subsection.

D. Data-to-insight Time

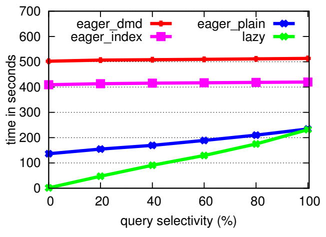
To vary the selectivity of the queries on the entire data space and get proportional amounts of data being selected for various selectivity levels, one need to have data that is uniformly distributed over different values of the metadata attributes. This might not be usually the case with a real dataset, as it is not with the INGV dataset. Not every station has uniformly distributed data over a specific time period. However after some digging into we have seen that one of the stations called “FIAM” has somewhat equally distributed data over its associated files. Thus, we have created another dataset (to be called FIAM dataset) which is roughly quarter of the size of the original dataset (i.e. it spans 3 years again but contains data from one station only). We still preserve the scale factoring (i.e. the entire FIAM dataset is sf-27). We use this dataset throughout the rest of the experiments.

We remove all selection predicates and projections from the queries, except the range predicate on the time for each query, which we will vary to change query selectivity over data space.

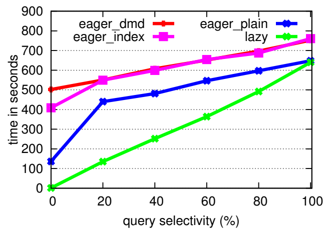
Figure 8 shows the data-to-insight time over 4 different loading approaches for sf-1 and sf-27 varying the query selectivity for running T_4 and T_5 queries. Queries are run as the first query after the preparatory effort (i.e. data-to-insight time). Apparently, results for 0% selectivity is the time spent during the preparation of different loading approaches. Additionally, T_2 and T_3 queries show similar patterns to T_5 query, so we omitted their graphs for space limitations.

The T_4 query performance with Eager_index and Eager_dmd show the same pattern with difference of index building time, whereas the curves completely align for T_5 query – except at the point of 0% as expected. This is because T_5 query on Eager_index has to compute the DMd as extra. Similarly T_5 on Eager_plain has to compute the DMd too, but without making use of the foreign key indexes. That’s why, it takes a much longer running time. Eager_plain curves lying always under the Eager_dmd curves apparently show us computing DMd without the indexes is faster than constructing indexes plus computing DMd. But, of course Eager_dmd would easily benefit if we have more number of queries.

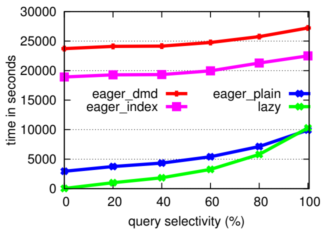
The running time of a query loaded sf-27 database with Lazy approach goes up more steeply towards the high selectivity levels. These are due to the fact that the main memory is becoming the bottleneck with the high selectivity because the entire dataset has to be loaded lazily during the query execution. Moreover, independent of the scale factor, the entire dataset is processed without any join indexes on Lazy (unlike the Eager_index and like the Eager_plain). This explains why



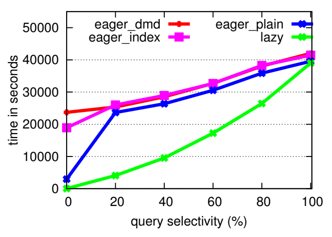
(a) T_4 query, sf-1



(b) T_5 query, sf-1



(c) T_4 query, sf-27



(d) T_5 query, sf-27

Fig. 8. First query performance for various query selectivity levels. All experiments use the FIAM dataset.

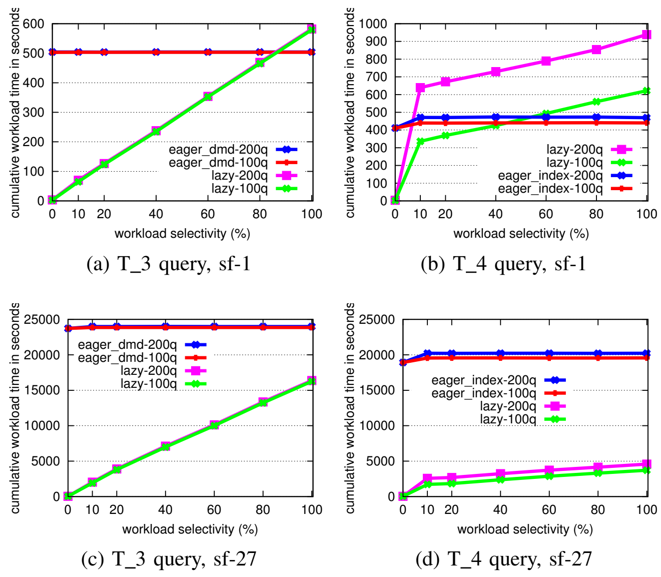
Lazy curve end up nearly at the same running time with that of Eager_plain. So, Lazy behaves similar to Eager_plain for high selectivity levels. This justifies that our approach is not designed for high selectivity levels.

The interesting point is, even firing a first query with 100% selectivity, Lazy still provides a shorter data-to-insight time than Eager_index and Eager_dmd approaches. With the rest of the selectivity levels, it has even shorter data-to-insight time. However, since the actual first query running time is shorter for Eager_index and Eager_dmd, this gives them advantage over a workload of queries, the effect of which we measure in the next subsection.

E. Workload of Queries

We use the same queries from previous experiments with a fixed selectivity of 2.5%. We vary the workload selectivity (i.e. data space covered by workload divided by data space) between 0 and 100%. Workloads has 100 and 200 queries, to see how number of queries affect the cumulative performance as well. The workload queries are randomly distributed over the workload space and we make sure that the workload space is fully covered. As previous experiments, the workload is run after the preparation effort. Therefore, 0% selectivity represents the time spent in preparation. Figure 9 shows total workload execution times over 2 different loading approaches (Lazy and the best of 3 eager approaches depending on the query type) for sf-1 and sf-27 for running T_3 and T_4 queries. T_5 and T_2 query has a similar pattern as T_3, so we present T_3 query for these experiments. Furthermore, we limit the size of the recycler cache holding the lazily loaded files to the size of main memory.

The eager approach has very similar workload performance over various workload selectivity. For T_3 query it just returns an item from the materialized view already computed. For T_4 query, it takes advantage of the selection predicate on



(a) T_3 query, sf-1 (b) T_4 query, sf-1

(c) T_3 query, sf-27 (d) T_4 query, sf-27

Fig. 9. Workload performance for various workload selectivity levels. All experiments use the FIAM dataset.

the actual data to keep a low cumulative workload time. Whereas our approach provide significantly better workload performance for low workload selectivities (roughly 5 times faster for 20% on sf-27).

As we have observed on results in Figure 8 that computing DMd without the indexes is faster than constructing indexes plus computing DMd. However, here we see that T_3 query running on sf-1 with Lazy loading approach, for 100% workload selectivity, the cumulative workload time is greater than Eager_dmd loading time (∼500s). This is due to the parallelization strategy being static, as mentioned in Section V. If we increase the scale factor to sf-27, for 100% workload selectivity, the Lazy approach cumulative workload time is a lot smaller than Eager_dmd loading time. This is because with the sf-27 queries having the same query selectivity with sf-1 queries, we load much more number of mSEED files during the query execution of Lazy. Hence this leads to a better parallelization over files. The same reason also causes the cumulative workload time of T_4 queries on sf-1 database with Lazy approach to quickly get closer to and even over the Eager_index workload times towards a higher workload selectivity. This is through running time of each individual query being increased in the workload. At the same time, this explains why Lazy already loses advantage with only 200 queries. As a result Lazy does not help much with smaller scale factors, unless the static parallelization strategy is left.

The interesting point is that, with data getting bigger our approach becomes more advantageous, even with 100% workload selectivity. As expected, increasing the number of queries benefit the eager approach. However, the benefit degrades towards a higher scale factor. For T_4 query on sf-27 database, the number of queries per workload can be increased orders of magnitude, while Lazy still having better workload performance than Eager_index.

In general, our approach provides better performance with smaller query and workload selectivity levels. Query selectivity has a bigger effect on the overall performance. Keeping the query selectivity lower in most of the queries provides a better workload performance, even with 100% workload selectivity.

VII. Related work

To the best of our knowledge, we have not seen any work targeting the “chunked data” to reduce the data preparation costs – though, there is chunked/blocked indexing structures (e.g. zone maps, min-max indices, etc.), however they add up to the data preparation effort. In general, the problem of exploring external data and data ingestion in particular, has received significant attention from the database community.

Data exploration typically involves a lengthy sequence of queries which dynamically adapts based on how the scientist interprets the data [23]. Although not all work on data exploration focuses on data-to-insight time (e.g. [24]), it is one of the key points of interactive data exploration. Beginning with this motivation, several in-situ processing papers have come out [15], [25], [26], [27] to provide dynamic and selective load of data from a file with zero initialization overhead and performance benefits for subsequent queries. However, external tables and the provided in-situ processing techniques are for single file accesses and/or does not exploit the chunked nature of the data if any, whereas we work on the level of file repositories (or in general chunked data). Hence, the approaches are orthogonal and even complementary (e.g., a NoDB-like in-situ processing approach can be incorporated into our system as an accessor for a single chunk, in order to provide sub-chunk access granularity).

Instant loading [28] is a CSV loading approach that allows scalable loading at wire speed. This is achieved by a synchronization-free task-parallelization (through chunking) and data-parallelization (through SIMD instructions) of parsing, deserialization, and input validation. The addressed problem is fundamentally different. Since we do not touch the internal mechanisms of actual physical loading process, but only change the timing of it, we still need to do parsing, deserialization and input validation.

Abouzied et al in [29] present a system that fills a database with external data in an access-driven fashion. As users' Hadoop jobs access and parse the data for processing, they piggyback on Hadoop jobs and load the data incrementally that is being accessed. However, the user, after copying into HDFS, has to manually specify in the Hadoop job the part or chunks of the dataset he wants to process, because no metadata is taken into account and exploited.

Our approach looks attractive for distributed setups. In principle, our approach could be merged with the MapReduce paradigm and can be implemented in a system like Hive [30] or other SQL-on-Hadoop systems. Firstly, such systems might assume a common-format but no chunking of data, which is not always appropriate for our target domains (e.g. scientific data processing). Secondly, we believe that this problem is orthogonal since any kind of scale-out can happen on top of single nodes that apply our approach.

Scientific middleware solutions, where databases are used only for metadata querying and then middleware application finds and opens the resulting files for further analysis, came onto stage to help with standardization of file structures and the metadata management (e.g., [8], [9], [10]). Although they exploit metadata by querying it explicitly, they do not offer in-database processing of the actual data. SciQL [31] on MonetDB and SciDB [32] are for processing of array data. Query processing and optimization exploit array semantics. However, loading of external data is an up-front step.

VIII. Future Work

While our prototype is functional, useful and available in open source, there is still potential for improvement. Let us, therefore, discuss future work before concluding.

Other Sources. We designed our system to be extensible to other kinds of data sources (e.g. image and video datasets, datasets of other scientific domain like astronomy, life sciences, etc.). As future work, we can go further, since much "big" data is not available in files but behind other kinds of interfaces. A particularly interesting interface is HTTP which could open up document-oriented databases such as Redis or web-based APIs like Flickr. Another useful interface is HDFS which holds much of today's "big" data.

Smarter Caching. To provide good performance for repetitive accesses to a data chunk, we strongly rely on MonetDB's existing intermediate caching component (the Recycler [6]). However, this component implements a plain LRU replacement policy and only indexes cached results when they are used for (hash-)joins. By extending the Recycler with a more sophisticated cost model, replacement and indexing decisions could be driven by loading costs as well as usage statistics.

Approximative Query Answering. One of the negative aspects of the lazy loading is that it shifts costs from preparation time to query time. When many chunks are selected, this can lead to unacceptable waiting times: a user might be willing to accept eight hours of preparation during the night rather than 15 minutes while waiting for a query result. To mitigate this problem, our approach can be combined with techniques of approximative query answering such as sampling.

IX. Summary and conclusions

With the increasing volume of "big" datasets, data preparation (storage, integration, indexing, ...) is becoming increasingly costly. To address this problem, we developed a data management system that limits the preparation to the data deemed relevant by the user. We use the user's queries to determine which data items are relevant and the metadata about *semantic chunks* of data to load chunks that contain data items of interest on-demand. To create the illusion of a fully populated database, we make each of the components of an existing data management system (MonetDB) *partial-loading aware*: we extended the query plan generator, the optimizers as well as the kernel itself. The result is a system that provides a unified view on internal as well external data and exploits the characteristics of the access paths where possible. While this approach still leaves potential for improvement using orthogonal techniques such as sophisticated cost-models or parallelization strategies, we achieve orders of magnitude lower data preparation times at competitive query evaluation times for a real-life scientific data management application.

References

[1] J. P. Dijcks, "Oracle: Big data for the enterprise," *White Paper*, 2012.

[2] J. Gray *et al.*, "Scientific Data Management in the Coming Decade," *SIGMOD Record*, vol. 34, no. 4, 2005.

[3] T. Jörg and S. Deßloch, "Towards generating etl processes for incremental loading," in *IDEAS 2008*. ACM, pp. 101–110.

[4] S. Idreos, M. L. Kersten, and S. Manegold, "Database Cracking," in *Proc. CIDR*, Asilomar, CA, USA, January 2007.

[5] J. Zhou, P.-A. Larson, J. Goldstein, and L. Ding, "Dynamic materialized views," in *ICDE*, 2007, pp. 526–535.

[6] M. Ivanova, M. Kersten, N. Nes, and R. Gonçalves, "An Architecture for Recycling Intermediates in a Column-store," in *SIGMOD*, 2009.

[7] A. Ailamaki, V. Kantere, and D. Dash, "Managing scientific data," *Commun. ACM*, vol. 53, no. 6, pp. 68–78, Jun. 2010.

[8] E. Stolte *et al.*, "Scientific data repositories: Designing for a moving target," in *SIGMOD 2003*.

[9] P. Baumann *et al.*, "The multidimensional database system RasDaMan," *SIGMOD Rec.*, vol. 27, no. 2, pp. 575–577, 1998.

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

[11] A. Shoshani *et al.*, "Characteristics of scientific databases," in *VLDB*. Morgan Kaufmann Publishers Inc., 1984, pp. 147–160.

[12] *Standard for the Exchange of Earthquake Data*. IRIS, February 1988.

[13] Y. Kargin, H. Pirk, M. Ivanova, S. Manegold, and M. Kersten, "Instant-On Scientific Data Warehouses — Lazy ETL for Data-Intensive Research," in *BIRTE*, 2012.

[14] J. D. Ullman, *Principles of database systems*, 1985.

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

[16] M. Ivanova, M. Kersten, and S. Manegold, "Data vaults: A symbiosis between database technology and scientific file repositories," in *SSDBM 2012*.

[17] M. Ivanova, M. Kersten, S. Manegold, and Y. Kargin, "Data vaults: Database technology for scientific file repositories," *CiSE*, vol. 15, no. 3, pp. 32–42, 2013.

[18] J. Zhou, P.-A. Larson, J. Goldstein, and L. Ding, "Dynamic materialized views," in *ICDE 2007*. IEEE.

[19] G. Luo, "Partial materialized views," in *ICDE 2007*. IEEE.

[20] (2015) MonetDB, Column-store Pioneers. www.monetdb.org.

[21] G. Graefe, *Encapsulation of parallelism in the Volcano query processing system*. ACM, 1990, vol. 19, no. 2.

[22] "The Mini-SEED Software Library, libmseed." 2015.

[23] S. Idreos *et al.*, "Here are my data files. here are my queries. where are my results?" in *CIDR 2011*.

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

[25] S. Blanas, K. Wu, S. Byna, B. Dong, and A. Shoshani, "Parallel data analysis directly on scientific file formats," in *SIGMOD 2014*.

[26] Y. Cheng and R. Florin, "Parallel in-situ data processing with speculative loading." SIGMOD, 2014.

[27] M. Karpathiotakis *et al.*, "Adaptive query processing on raw data," in *VLDB*, 2014.

[28] T. Mühlbauer, W. Rödiger, R. Seilbeck, A. Reiser, A. Kemper, and T. Neumann, "Instant loading for main memory databases," *VLDB*, 2013.

[29] A. Abouzied *et al.*, "Invisible loading: Access-driven data transfer from raw files into database systems," ser. EDBT '13.

[30] A. Thusoo, J. S. Sarma *et al.*, "Hive: a warehousing solution over a map-reduce framework," *VLDB*, 2009.

[31] Y. Zhang *et al.*, "SciQL: bridging the gap between science and relational DBMS," in *IDEAS '11*. ACM.

[32] M. Stonebraker *et al.*, "Requirements for Science Data Bases and SciDB," in *CIDR*, 2009.