Instant-On Scientific Data Warehouses

Lazy ETL for Data-Intensive Research

Yağız Kargın, Holger Pirk, Milena Ivanova, Stefan Manegold, Martin Kersten

Centrum Wiskunde & Informatica (CWI), Amsterdam, The Netherlands

Abstract. In the dawn of the data intensive research era, scientific discovery deploys data analysis techniques similar to those that drive business intelligence. Similar to classical Extract, Transform and Load (ETL) processes, data is loaded entirely from external data sources (repositories) into a scientific data warehouse before it can be analyzed. This process is both, time and resource intensive and may not be entirely necessary if only a subset of the data is of interest to a particular user. To overcome this problem, we propose a novel technique to lower the costs for data loading: *Lazy ETL*. Data is extracted and loaded transparently on-the-fly only for the required data items. Extensive experiments demonstrate the significant reduction of the time from source data availability to query answer compared to state-of-the-art solutions. In addition to reducing the costs for bootstrapping a scientific data warehouse, our approach also reduces the costs for loading new incoming data.

1 Introduction

Thirty years of database research knowledge is slowly finding its way into the domain of science [7]. This trend is driven by the need to handle the large amounts of data that are the result of increasingly automated acquisition of scientific data. Especially life and earth sciences obtain more and more data ever faster and faster. The capacity to sequence genomes, e.g., has been outpacing Moore's Law in the last years and is expected to continue to do so [24].

In addition to the increasing data volume, the character of science itself changes. In the past, researchers used to form hypotheses and validate them using experiments. Today they first run cheap, high throughput experiments and mine the results for "interesting" knowledge. This became known as eScience and culminated in the formulation of *the fourth paradigm* [7]. In its essence, this process (see Figure 1) is similar to that which drives business intelligence.

It is, therefore, natural to consider the techniques that helped meet the requirements of business intelligence to help scientists in their daily work. A significant amount of research has been done on various aspects of eScience. This covers problems of querying [18], mining [13] and visualization [19]. However, all of these assume that the underlying data is readily available in the scientist's database. This is generally not the case.

Instead, input data is usually collected in domain-specific files and repositories (i.e., semi-structured collections of files). The need to physically load all

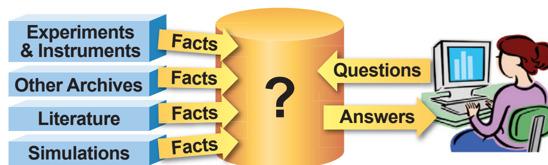

Fig. 1. EScience Knowledge Management (taken from [7])

data into a data warehouse system before analysis forms a burden that domain scientists are reluctant to accept. This burden is exacerbated by the fact that few scientists have access to data integration specialists. In general, scientists have to take care of their data management needs themselves. Thus, they abstain from "superior solutions" that require significant data management knowledge. Our objective is to provide eScientists with a data ingestion system that is easy to use, extensible and scalable to many terabytes of data. To this end, we present a query-driven, on-demand Extract, Transform & Load (ETL) system that minimizes the burden of data ingestion.

To achieve the necessary scalability, our approach limits the initial loading to the metadata of the input files. The actual data is extracted, transformed and loaded transparently at query time for the files containing the data that the queries require. In addition to reducing the cost for bootstrapping a scientific data warehouse for a new file repository, our approach also makes updating and extending a warehouse with modified and additional files more efficient.

We implemented our novel *Lazy ETL* system in MonetDB [1], an open-source column-store Database Management System (DBMS). To showcase our solution, we use typical analysis tasks on seismological sensor data that is available in mSEED files and a file repository as data source, which is one of the kinds of source datastores that ETL processes deal with [23]. Extensive experiments demonstrate the significant reduction of the overall time from source data availability to query answer compared to state-of-the-art solutions that require the ETL task to be completed for the entire data before running queries. Even though we focus on a particular scientific use case, we believe that other cases, like classical business ETL, can benefit from a similar approach, as also mentioned by Dayal et al. [5] and Haas et al. [6].

We organized the rest of this paper as follows: In Section 2 we describe the state of the art in data ingestion and discuss its shortcomings with particular focus on scientific data management. In Section 3 we present our architecture for overcoming these shortcomings without losing the merits of traditional ETL. In Section 4 we evaluate our solution in a scientific case, and we draw conclusions in Section 5.

2 State of the Art in Data Ingestion

As explained, the data ingestion needs of eScientists are very similar to those of business analysts. We, therefore, describe the state of the art of data inges-

tion in business data warehouses before we explore the transferability into the eScience domain. EScientists load data from sensor data files much like business warehouses load data from Online Transaction Processing (OLTP) systems.

The most popular setup to serve transactional and analytical applications is the following [4]: a row oriented operational DBMS for the OLTP-load and a Data Warehouse for the analytical needs. New data enters the operational system as it occurs and is loaded into the Warehouse in intervals using an *ETL* process [14]. This, however, has several drawbacks:

1. The data that has not been transferred to the OLAP-store yet, will not appear in the aggregated results, which renders the OLAP-store constantly out of date [17].
2. All data has to be held redundantly which increases the costs for hardware acquisition and maintenance [14].
3. The update process has to be maintained and run periodically to keep the OLAP-store reasonably up to date. Since this process can be complicated the added costs in hardware and personnel can be high [14].

The costs may increase even further with the complexity of the management's requirements. A common requirement that is especially interesting is real time reporting. Vendors support this through means of *Active Warehousing*.

2.1 Active Warehousing

To increase the efficiency of business operations it is often required to do analytics in a relatively short period of time (an hour or even minutes). This kind of *Operational Reporting* [10] is a trend that has been recognized by vendors [3]. They aim at supporting it by means of *Active Warehousing*: Shortening of the update interval. This reduces the deviance of the aggregates from the real, transactional data and therefore allows almost real time reporting. It does however chronically increase the load on both the transactional and the analytical database. The transactional database has to handle additional extracts which cannot, as is common in traditional warehousing, be scheduled in the downtime of transactional operations but have to be executed concurrently to the transactional load.

2.2 Lazy Aggregates

The update interval in Active Warehouses is shorter than in traditional warehouses but still a constant. The deviance between the real data is therefore undetermined because it may be changed arbitrarily by a transaction unless special restrictions are implemented. A possibility to limit this deviance is provided by a technique known as *Lazy Aggregates* [15]. The warehouse update is not triggered after a given interval but when the deviance exceeds a predefined threshold. This assumes that it is significantly faster to calculate the deviance that is induced by a processed transaction than to run the update. Depending on

the aggregation function calculating the deviance without calculating the value can be costly or even impossible (e.g., for holistic functions). In that case this approach fails to yield any benefit.

2.3 Scientific Data Ingestion

We are not the first to recognize the need for solutions that support scientific data ingestion. In particular, the problem of loading data from external files has received significant attention from the database community. The SQL/MED standard (Management of External Data) [21] offers an extension to the SQL standard that simplifies the integration of external files into a relational schema. It allows the SQL-server to control referential integrity, authorization, and recovery for data in external files. However, it still puts the burden of managing the files on the user. Using the Oracle Database File System (DBFS) [16], files with unstructured data are transparently accessible as BLOB-attributes of relational tuples. But DBFS provides no structured view to external file contents. Similarly, there are also well-established techniques like external tables in any major commercial system. These provide us access to data in external sources as if it were in a table in the data warehouse. However, external tables require every query to access the entire data as opposed to Lazy ETL accessing only the data that queries require. DBMS query processing over flat data files is proposed by Idreos et al. [8]. They provide query-driven on-demand loading as we do. However, they limit the format to those that map straightforward to relational tables, such as CSV and tables in Flexible Image Transport System (FITS). Consequently they cannot handle more complex file formats that are common in eScience applications like MiniSEED (mSEED) or Geo Tagged Image File Format (GeoTIFF). These files contain complex schemas of different types of objects and even pointers that should be resolved to foreign key constraints when loading. The need for symbiosis of databases and file systems for eScience is in the core of the Data Vault concept presented in [11]. Our work can be seen as a further development of the idea of just-in-time access to data of interest integrated with the query processing.

Hence, none of the above provides an adequate solution to the data ingestion problem in eScience. In the following we present our approach to the problem: *Lazy ETL*. While targeted at scientific use cases, we believe that our solution is generally applicable to ingestion problems that involve (repositories of) external files.

3 Lazy ETL

Traditional ETL relies on proactively filling the data warehouse with data. However, a user may only need a subset of the available database. In the extreme case, he might only want a fast answer to one ad-hoc query. Nevertheless, he still has to wait for the lengthy *eager* loading of all the input data. To mitigate

this problem, only the "interesting" part of the data could be loaded. However, without knowledge of the workload it is unclear which subset of data is interesting. Hence, traditional ETL processes load and pre-aggregate all data that can possibly be queried into the data warehouse.

Breaking with the traditional paradigm, we consider ETL as part of the query processing. This allows us to run ETL only when required and only for the data that is required by a query. We call this approach *Lazy ETL*. To implement Lazy ETL, we create a virtual warehouse that is filled with data on demand. While executing queries, required data that is not yet present in the warehouse is loaded. This does not only facilitate the initial loading, but also the refreshing of the data warehouse contents. After the initial loading, refreshments are handled inherently when the data warehouse is queried.

In the following we illustrate how to replace a traditional ETL process with our lazy ETL scheme without loosing the benefits of the traditional approach. We visit each step of the lazy ETL process to discuss the further details.

3.1 Lazy Extraction

Extraction is the process of making the input data available for subsequent transformation and loading. Traditionally all data items that may be needed during transformation are extracted from the input data sources. In the typical ETL case with OLTP systems as data sources, structured queries may be used to limit the extracted data. This is particularly useful for incremental updates after the initial loading [22]. For flat file repositories, however, such optimization opportunities are limited to whatever the respective file format library provides. This intensifies the challenge of incremental updates and provides additional opportunities for lazy extraction.

Metadata Without any knowledge about the input data files, all available files have to be considered "relevant" for a given query. Fortunately, many scientific file formats define the notion of *metadata*. Metadata is data that provides some insight into the content of a file but is still cheap to acquire. In an eScience scenario this covers data such as the time, the sampling rate or the position of a sensor. The costs to acquire such metadata is usually many orders of magnitude lower than loading the actual data. Metadata may even be encoded in the filename which would allow extraction without even opening the file. Therefore, we consciously decided to load the metadata eagerly. This initial investment will be amortized with the first query and provides the user with some overview of what to expect in the actual data. Although the definition of metadata might differ with file format, use case and system performance, in most of the cases it is relatively straightforward to decide which data to load eagerly. If no metadata is available for a given data source it may prove beneficial to manually define it.

Actual Data We call all data that is not metadata actual data. In the eScience domain actual data items usually describe individual data points and come with, e.g., a timestamp and one or many measured values. In practice, we expect the

majority of the data items to be actual data. Actual data is, therefore, subject to lazy ETL. Although this query-driven on-demand extraction might cause overhead in source datastores, it is still only for a smaller set of data required for a query on the contrary to the eager incremental updates.

To select the files to load, the selection predicates on the metadata are applied. Once this part of the plan is executed, a rewriting operator is executed. This operator uses plan introspection and modification that are provided by the MonetDB system to replace all references to relational tables with operators that load the necessary files. This allows us to seamlessly integrate lazy extraction with query evaluation.

3.2 Lazy Transformation

An ETL process might contain transformations for different tasks, such as pivoting, one-to-many mappings, schema-level transformations etc. [22] [23]. Most of these transformations (e.g., projections, joins, etc.) can be provided by a relational query processor. It is therefore common practice to express these transformations in SQL. Transformations that use richer data mining can be potentially handled by user defined functions. This makes our approach similar to industrial common practice ELT (Extract, Load & Transform), but ELT does not have the laziness concept.

In our approach we implement all necessary transformations as relational views on the extracted data. This happens transparent to the user and has a number of benefits:

- It naturally supports lazy processing of the transformations because view definitions are simply expanded into the query. Only the minimal amount of needed data items from the sources are transformed.
- It allows the transformations to benefit from query optimization. These may even be based on information that is not known at "eager" load time (e.g., data distribution or selectivities).
- It provides an inherent means of data provenance since the transformations are still visible at query time.

3.3 Lazy Loading

Loading in ETL is the storing of the extracted and transformed data into the internal data structures of the data warehouse. This is largely done for performance reasons and is, thus, treated as an optimization in our system. From a query processing point of view, materialization of the transformed data is simply caching the result of the view definition. Our target system, MonetDB, already supports such caching in the form of *intermediate result recycling* [12].

Whilst not part of our current system, we consider integration of lazy ETL with intermediate result recycling a promising optimization and part of future work. This will make lazy ETL even lazier. Note that caching comes with the challenge of keeping the cache up to date.

4 Evaluation

To evaluate our approach, we have chosen seismology as the scientific domain for our use case. Our choice is mainly driven by a tight cooperation with project partners, seismologists from the Royal Dutch Meteorological Institute (KMNI). We strongly believe that other scientific domains have similar use cases, and that our techniques and result can thus be generalized to other science fields.

4.1 Data Source

Seismology is a scientific domain where huge amounts of data is generated with ever lasting movements of the Earth's surface. In seismology, the Standard for the Exchange of Earthquake Data (SEED)[2] is the most widely used standard file format to exchange waveform data among seismograph networks (e.g., transferring data from a station processor to a data collection center). A SEED volume has several ASCII control headers and highly compressed data records, i.e., the waveform time series. The control headers keep the metadata, which consists of identification and configuration information about the data records (i.e., actual data). In our experiments we use mSEED files, which contain less metadata. mSEED files are kept in large remote file repositories with direct FTP access [20].

4.2 Data Warehouse Schema

The normalized data warehouse schema for our use case is derived straightforwardly from the mSEED file format. An mSEED *file* contains multiple mSEED *records* (about 35 records per mSEED file on average in our data collection). An mSEED record contains the sensor readings over a consecutive time interval, i.e., a timeseries of about 3500 values on average in our data collection. Consequently, the normalized data warehouse schema — as given in Listing 1 — consists of three tables and one view. Tables `files` and `catalog` hold the metadata per mSEED file and mSEED record, respectively, while table `data` stores the actual sensor data. Each mSEED file is identified by its URI (`file_location`), and contains the metadata describing the sensor that collected the data (`network`, `station`, `location`, `channel`) as well as some technical data characteristics (`dataquality`, `encoding`, `byte_order`). Each record is identified by its (record) sequence number (unique per file), and holds metadata such as `start_time`, sampling rate (`frequency`), and number of data samples (`sample_count`). The `data` table stores the timeseries data as (`timestamp`, `value`) pairs. For user convenience, we define a (non-materialized) view `dataview` that joins all three tables into a (de-normalized) "universal table".

4.3 Sample Workload

Our sample workload consists of the eight queries presented in Listing 2 that reflect characteristic data analysis tasks as regularly performed by seismologists

Listing 1. Data Warehouse Schema

```
CREATE SCHEMA mseed;

CREATE TABLE mseed.files (
  file_location STRING,         dataquality   CHAR(1),
  network       VARCHAR(10),    station       VARCHAR(10),
  location      VARCHAR(10),    channel       VARCHAR(10),
  encoding      TINYINT,        byte_order    BOOLEAN,
  CONSTRAINT files_pkey_file_loc PRIMARY KEY (file_location)
);

CREATE TABLE mseed.catalog (
  file_location STRING,         seq_no        INTEGER,
  record_length INTEGER,        start_time    TIMESTAMP,
  frequency     DOUBLE,         sample_count  BIGINT,
  sample_type   CHAR(1),
  CONSTRAINT catalog_file_loc_seq_no_pkey PRIMARY KEY (file_location, seq_no),
  CONSTRAINT cat_fkey_files_file_loc
    FOREIGN KEY (file_location)
      REFERENCES mseed.files(file_location)
);

CREATE TABLE mseed.data (
  file_location STRING,         seq_no        INTEGER,
  sample_time   TIMESTAMP,      sample_value  INTEGER,
  CONSTRAINT data_fkey_files_file_loc
    FOREIGN KEY (file_location)
      REFERENCES mseed.files(file_location),
  CONSTRAINT data_fkey_catalog_file_loc_seq_no
    FOREIGN KEY (file_location, seq_no)
      REFERENCES mseed.catalog(file_location, seq_no)
);

CREATE VIEW mseed.dataview AS
SELECT
  f.file_location, dataquality, network, station, location, channel, encoding,
  byte_order, c.seq_no, record_length, start_time, frequency, sample_count,
  sample_type, sample_time, sample_value
FROM mseed.files AS f
  JOIN mseed.catalog AS c
    ON f.file_location = c.file_location
  JOIN mseed.data AS d
    ON c.file_location = d.file_location AND c.seq_no = d.seq_no;
```

while hunting for "interesting seismic events". Such tasks range from finding extreme values over *Short Term Averaging* (*STA*, typically over an interval of 2 seconds) and *Long Term Averaging* (*LTA*, typically over an interval of 15 seconds) to retrieving the data of an entire record for visualization and visual analysis. Query 1 finds the maximum values (amplitudes) for a given channel (`BHN`) per station in the Netherlands (`NL`) during the first week of June 2010. Queries 2 and 3 compute the short term average over the data generated at Kandilli Observatory in Istanbul (`ISK`) via a specific channel (`BHE`). The difference between Query 2 and 3 is that Query 3 has an additional range predicate on the `start_time` of the records that is semantically redundant, but might help the query optimizer to pre-filter on the (small) `catalog` table before evaluating the predicate on `sample_time` on the (huge) `data` table. Query 4 computes the long term average over the data from the same station as Queries 2 and 3,

Listing 2. Sample Queries

```
-- Query 1
SELECT station, MAX(sample_value)
FROM mseed.dataview
WHERE network = 'NL'
  AND channel = 'BHN'
  AND start_time > '2010-06-01T00:00:00.000'
  AND start_time < '2010-06-07T23:59:59.999'
GROUP BY station;

-- Query 2
SELECT AVG(sample_value)
FROM mseed.dataview
WHERE station = 'ISK'
  AND channel = 'BHE'
  AND sample_time > '2010-01-12T22:15:00.000'
  AND sample_time < '2010-01-12T22:15:02.000';

-- Query 3
SELECT AVG(sample_value)
FROM mseed.dataview
WHERE station = 'ISK'
  AND channel = 'BHE'
  AND start_time > '2010-01-12T00:00:00.000'
  AND start_time < '2010-01-12T23:59:59.999'
  AND sample_time > '2010-01-12T22:15:00.000'
  AND sample_time < '2010-01-12T22:15:02.000';

-- Query 4
SELECT channel, AVG(sample_value)
FROM mseed.dataview
WHERE station = 'ISK'
  AND sample_time > '2010-01-12T22:15:00.000'
  AND sample_time < '2010-01-12T22:15:15.000'
GROUP BY channel;

-- Query 5
SELECT channel, sample_time, sample_value
FROM mseed.dataview
WHERE station = 'ISK'
  AND sample_time > '2010-01-12T22:15:00.000'
  AND sample_time < '2010-01-12T22:18:00.000';

-- Query 6
SELECT station, MIN(sample_value), MAX(sample_value)
FROM mseed.dataview
WHERE network = 'NL'
  AND channel = 'BHZ'
GROUP BY station;

-- Query 7
SELECT sample_time, sample_value
FROM mseed.dataview
WHERE seq_no = 1
  AND file_location =
      '/.../knmi/ORFEUS/2010/152/NL_HGN_02_LHZ.2010.152.16.47.34.mseed';

-- Query 8
SELECT sample_time, sample_value
FROM mseed.dataview
WHERE seq_no = 1
  AND file_location =
      '/.../knmi/ORFEUS/2010/158/NL_OPLO_04_BHN.2010.158.09.26.51.mseed';
```

but now over all available channels. Query 5 retrieves a piece of waveform at a given location, e.g., to visualize the data around a seismic event detected via Queries 2, 3, 4. Query 6 is similar to Query 1, but calculates both minimum and maximum for a given channel (`BHN`) per station in the Netherlands (`NL`) without restricting the time period. Queries 7 and 8 retrieve the time series data of an entire record from a given file, e.g., for visualization, or further analysis by an external program. The difference between Query 7 and 8 is that Query 8 asks for a file that does not exist, i.e., yields an empty result.

4.4 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).

A copy of the ORPHEUS mSEED file repository is stored on a local server and accessible via NFS. The extraction of (meta)data from mSEED files is realized with the libmseed library [9].

We use MonetDB [1] as data warehouse, and extended it with our Lazy ETL techniques.

records per table			size				
`files`	`catalog`	`data`	mSEED	CSV	MonetDB	+keys	Lazy
5,000	175,765	660,259,608	1.3 GB	74 GB	13 GB	9 GB	10 MB
10,000	359,735	1,323,307,090	2.7 GB	148 GB	26 GB	18 GB	20 MB
20,000	715,738	2,629,496,058	5.5 GB	293 GB	50 GB	38 GB	36 MB

Table 1. Datasets and their sizes

We create 3 different datasets of increasing size by randomly selecting, respectively, 5000, 10000, and 20000 files of the 161329 files from year 2010. Table 1 lists some characteristics of the 3 datasets used. (The whole ORPHEUS repository holds more than 3.5 million files collected since 1988.)

We compare the following 3 different ETL approaches.

EagerExt refers to the traditional variant where an external stand-alone program reads the mSEED files and extracts the data. The transformation step involves solving a few schema- and value-level problems, since an mSEED file has somewhat different representations of the same data (e.g., not normalized). Then the program writes 3 CSV files, one for each table of our data warehouse schema. These files are then bulk-loaded into the data warehouse (using the `COPY INTO` SQL statement of MonetDB).

EagerSvr moves the functionality to read mSEED files into the DBMS server. We extended MonetDB with the required functionality to read mSEED files and extract their data into the tables of our data warehouse schema. Like EagerExt, EagerSvr still *eagerly* reads and loads the data from all given

mSEED files before querying can begin. However, it does not require to serialize the data into a CSV file and parse that again. Rather, the data from the mSEED files is directly loaded into the database tables inside the DBMS server.

Lazy refers to our new approach as presented in Section 3. Initially, the database server only extracts the metadata from all given mSEED files and loads it into tables `files` and `catalog`. Only during query evaluation the actual time series data is extracted and loaded at the granularity of an individual file into a temporary `data` table and required records are taken. For now, to keep the storage footprint of the database server at a minimum, the loaded data is discarded as soon and the query has been evaluated. While caching loaded data might avoid repeated loading of the same files, the chosen approach inherently ensures up-to-date data even if the original files are updated. A detailed study when and how to cache and update loaded data goes beyond the scope of this paper and is left for future work.

In order to analyze the costs and benefits of creating and using primary and foreign key constraints, we consider two variations of both EagerExt and EagerSvr. The first variant — *EagerExt-* / *EagerSvr-* — omits the primary and foreign key constraints, while the second variant — *EagerExt+* / *EagerSvr+* — creates the respective indexes after data loading, but before querying starts. The rationale is that creating primary and foreign key indexes, in particular for the foreign keys from `data` into both `catalog` and `files`, increases the data loading costs and storage requirements, but can speed up query processing. For Lazy, we do not build any indexes.

With the rather small `files` and `catalog` tables, primary and foreign key indexes do not show any significant performance advantage during query evaluation. For the `data` table, the foreign key indexes could only be built during query evaluation after the data is loaded on-the-fly. However, since creating a foreign key index basically means calculating the foreign key join, this does not yield any benefit.

4.5 Loading

Table 1 lists the characteristics of our three datasets as well as the size of the original mSEED files, the CSV files generated by EagerExt, the size after loading into MonetDB without primary and foreign key indexes, 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, serialization into a textual representation and explicit materialization of timestamps, the CSV files are much larger than the mSEED files.

An interesting point is that our lazy ETL approach is more space-efficient than eager approaches. The total amount of data stored in the data sources and the data warehouse is significantly less for the lazy case than for the eager cases.

Figure 2 breaks down the data ingestion costs into (a) extracting data from mSEED files to CSV files, (b) bulk loading data from CSV files into the DBMS

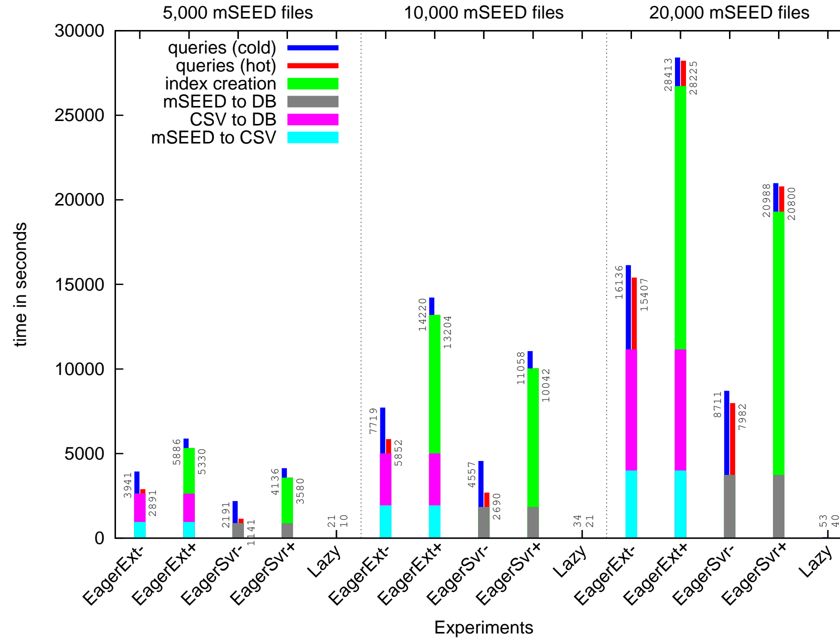

Fig. 2. Loading

(both for EagerExt), (c) loading the data directly from mSEED files into the DBMS (EagerSvr), and (d) creating primary and foreign key indexes. Additionally, Figure 2 shows the cumulative time for running all 8 workload queries 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).

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, EagerSvr is significantly faster than EagerExt, 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 data ingestion times for the Eager variants. The benefit of exploiting these indexes during query processing is visible. However, the difference is rather small, such that the investment pays off only after many queries.

4.6 Querying

Figures 3 through 5 show the individual times for all 8 queries as well as their cumulative execution time ("all") for the various cases discussed above. To accommodate widely spread execution times, we use a logarithmic scale for the y-axis. We see that for cold runs, Lazy consistently outperforms both Eager variants. With hot runs, Lazy falls slightly behind Eager in some cases due to

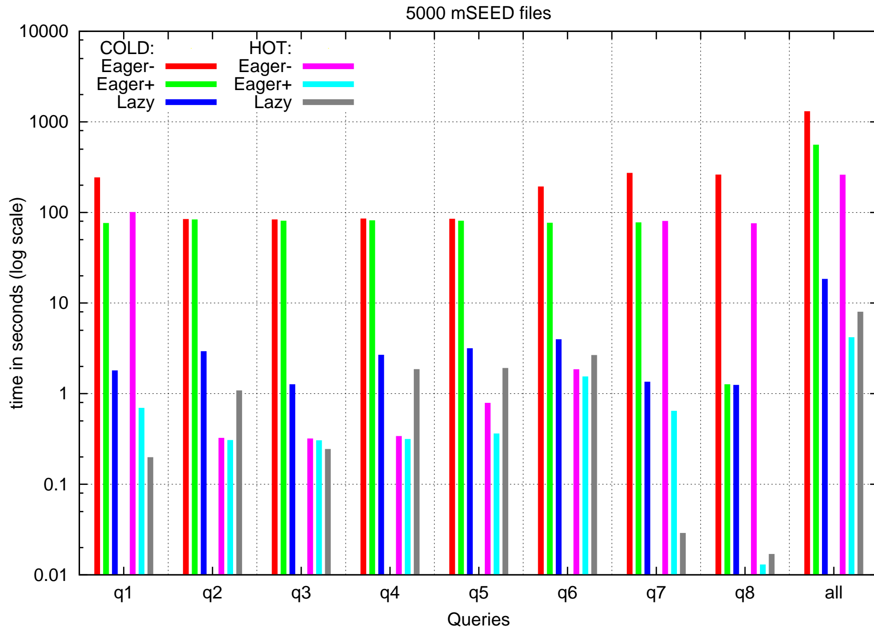

Fig. 3. Querying 5000 files

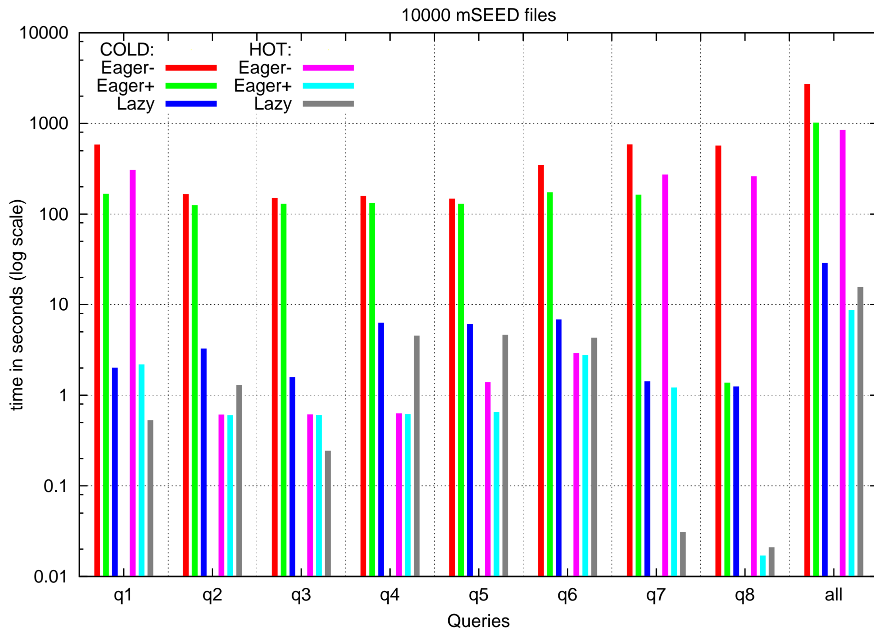

Fig. 4. Querying 10000 files

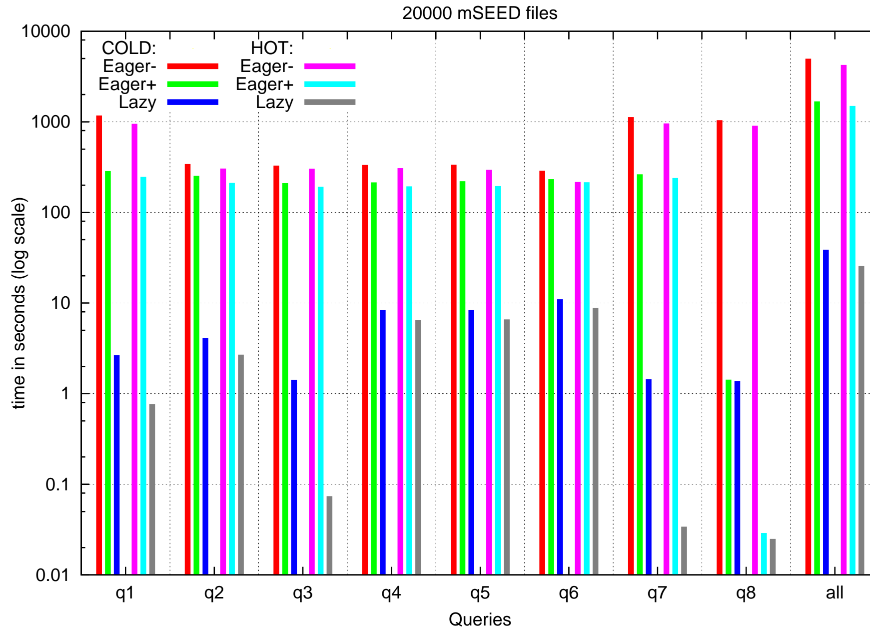

Fig. 5. Querying 20000 files

lazy ETL's on-the-fly reading of mSEED data files during execution as explained in section 3.

In Lazy, Query 3 is executed faster than Query 2 in all three datasets. This shows us that query performance of lazy ETL increases as the number of selections in the query increase, narrowing down the query. The execution times stay almost the same for eager variants, though. This is because an extra selection is introduced on the `catalog` table, although the size of result gets smaller. This also demonstrates that query performance of the lazy ETL process is dependent on how much actual data is required for the query answer. Intuitively, for the file repository case, it can range from loading no file at all (as for Query 8) to loading all the files in the worst case, where then the performance becomes similar to the initial loading of EagerSvr.

If we compare the total query times of 5000, 10000 and 20000 files, we see that query times in the eager case benefit from the indices less towards larger number of files. This is because the size of the indices is getting increasingly larger than the available main memory, which makes disk IO overhead increasingly dominant.

Running queries hot shows only a slight improvement in the performance for 20000 files, whereas it makes Eager approaches outperform the Lazy approach for 10000 files and even more for 5000 files. This is heavily due to the following reason. With the increase in dataset size, there is a significantly increased amount of data to deal with. Similarly, the lazy ETL process has to load more and more

files and ends up with more data in the data warehouse, but still small enough to handle conveniently. This shows that lazy ETL is less vulnerable to performance problems due to dataset sizes and can more easily scale to larger data sizes than eager ETL.

4.7 Discussion

We demonstrated the significant decrease in the time from source data availability to first query answer provided by the lazy ETL. Eager ETL can lead to better query performance after spending enough effort on initial loading. However, this results in data redundancy. Our approach also provides fresh actual data, although metadata might become stale. Typical ETL techniques (i.e., periodic and incremental ETL processes) can be employed in this case for the relatively small metadata only. Moreover, since we do not actually populate the data warehouse, the amount of data to deal with for query execution in the data warehouse tends to be relatively small. Furthermore, in the resumption problem of ETL, if an eager ETL process fails, this affects the overall system performance. On the other hand, if a lazy ETL process fails, only the query that triggered it is affected.

5 Conclusion

Scientific domains can benefit from techniques developed for business ETL. We proposed in this paper that the reverse is also correct. To make it possible, we presented lazy ETL processes that integrate ETL with query processing of data warehouses. Instead of actually populating the data warehouse, we temporarily made actual data queried accessible to it. This saved us from the burden of propagating updates for most of the data into the data warehouse, which has received a lot of research effort from the ETL community. We believe this contributes to the ETL research, at least in the case where the data sources are files.

Acknowledgments

This publication was supported by the Dutch national program COMMIT. The work reported here has partly been funded by the EU- FP7-ICT project TELEIOS.

References

1. MonetDB, Column-store Pioneers. www.monetdb.org.
2. *Standard for the Exchange of Earthquake Data*. Incorporated Research Institutions for Seismology, February 1988.

3. S. Brobst and A.V.R. Venkatesa. Active Warehousing. *Teradata Magazine*, 2(1), 1999.
4. S. Chaudhuri and U. Dayal. An overview of data warehousing and OLAP technology. *ACM Sigmod record*, 26(1):65–74, 1997.
5. U. Dayal, M. Castellanos, A. Simitsis, and K. Wilkinson. Data integration flows for business intelligence. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, pages 1–11. ACM, 2009.
6. L. Haas, M. Hentschel, D. Kossmann, and R. Miller. Schema and data: A holistic approach to mapping, resolution and fusion in information integration. *Conceptual Modeling-ER 2009*, pages 27–40, 2009.
7. A.J.G. Hey, S. Tansley, and K.M. Tolle. *The fourth paradigm: data-intensive scientific discovery*. Microsoft Research Redmond, WA, 2009.
8. S. Idreos, I. Alagiannis, R. Johnson, and A. Ailamaki. Here are my data files. here are my queries. where are my results? In *5th International Conference on Innovative Data Systems Research (CIDR)*, 2011.
9. Incorporated Research Institutions for Seismology. libmseed: The Mini-SEED Software Library, 2011.
10. B. Inmon. Operational and informational reporting. DM Review Magazine, 2000.
11. M. Ivanova, M.L. Kersten, and S. Manegold. Data vaults: A symbiosis between database technology and scientific file repositories. In *SSDBM*, pages 485–494, 2012.
12. M. Ivanova, M.L. Kersten, N.J. Nes, and R. Gonçalves. An Architecture for Recycling Intermediates in a Column-store. In *SIGMOD Conference*, pages 309–320, 2009.
13. S. Jaeger, S. Gaudan, U. Leser, and D. Rebholz-Schuhmann. Integrating protein-protein interactions and text mining for protein function prediction. *BMC bioinformatics*, 9(Suppl 8):S2, 2008.
14. M. Jarke, M. Lenzerini, Y. Vassiliou, and P. Vassiliadis. *Fundamentals of data warehouses*. Springer Verlag, 2003.
15. J. Kiviniemi, A. Wolski, A. Pesonen, and J. Arminen. Lazy aggregates for real-time OLAP. *Lecture notes in computer science*, pages 165–172, 1999.
16. K. Kunchithapadam, W. Zhang, et al. Oracle Database Filesystem. In *SIGMOD*, pages 1149–1160, 2011.
17. W.J. Labio, R. Yerneni, and H. Garcia-Molina. Shrinking the Warehouse Update Window. In *In Proceedings of SIGMOD*, pages 383–394, 1998.
18. J. López, C. Degraf, T. DiMatteo, B. Fu, E. Fink, and G. Gibson. Recipes for Baking Black Forest Databases - Building and Querying Black Hole Merger Trees from Cosmological Simulations. In *SSDBM*, pages 546–554, 2011.
19. P. Oldham, S. Hall, and G. Burton. Synthetic biology: Mapping the scientific landscape. *PLoS ONE*, 7(4):e34368, 04 2012.
20. ORFEUS. Seismology Event Data (1988 - now).
21. SQL/MED. ISO/IEC 9075-9:2008 Information technology - Database languages - SQL - Part 9: Management of External Data (SQL/MED).
22. P. Vassiliadis. A survey of extract–transform–load technology. *International Journal of Data Warehousing and Mining (IJDWM)*, 5(3):1–27, 2009.
23. P. Vassiliadis and A. Simitsis. Extraction, transformation, and loading. *Encyclopedia of Database Systems*, pages 1095–1101, 2009.
24. K.A. Wetterstrand. DNA sequencing costs: data from the NHGRI large-scale genome sequencing program. *Retrieved Available at: www.genome.gov/sequencingcosts. Accessed [2011-10-25]*, 2011.