

# ASSET Queries: A Set-Oriented and Column-Wise Approach to Modern OLAP

Damianos Chatziantoniou

Yannis Sotiropoulos

Department of Management Science and Technology,  
Athens University of Economics and Business (AUEB),  
Patission Ave, 104 34 Athens, Greece,  
{damianos, yannis}@aueb.gr

**Abstract.** Modern data analysis has given birth to numerous grouping constructs and programming paradigms, way beyond the traditional group by. Applications such as data warehousing, web log analysis, streams monitoring and social networks understanding necessitated the use of data cubes, grouping variables, windows and MapReduce. In this paper we review the associated set (ASSET) concept and discuss its applicability in both continuous and traditional data settings. Given a set of values  $B$ , an associated set over  $B$  is just a collection of annotated data multisets, one for each  $b \in B$ . The goal is to efficiently compute aggregates over these data sets. An ASSET query consists of repeated definitions of associated sets and aggregates of these, possibly correlated, resembling a spreadsheet document. We review systems implementing ASSET queries both in continuous and persistent contexts and argue for associated sets' analytical abilities and optimization opportunities.

**Keywords:** Associated sets, OLAP, Continuous queries, Spreadsheets.

## 1 Introduction

Today's complex world requires state-of-the-art data analysis over truly massive data sets. These data sets can be stored persistently in databases (possibly of different vendors) or flat files, or can be generated at real-time in a continuous, transient manner. Being able to easily express and efficiently evaluate integrated analytics over heterogeneous data sources is a major challenge in data management. In this paper we present the concept of associated set and review tools to express and engines to evaluate queries involving associated sets.

The ability to loop over the values of a domain and perform a task for each value is the main construct in programming languages and its presence leads to very strong theoretical results. Most database instructors explain initially the basic SQL statement (`select..from..where`) by using the procedural model, i.e. iteration. An associated set is simply a set of potential subsets of a data source  $S$ , one for each value  $b$  of a domain  $B$ , i.e.  $\{S_b: b \in B\}$ . An associated set instance (wlog often just called associated set) is a set of actual subsets of  $S$ . In our framework,  $B$  is usually a relation (the base relation), the data source  $S$  can be anything with a relational interface and an iterator

defined over it, and  $\theta$  is a defining condition that constraints (creates) the associated set instances. We claim that this simple approach: (a) generalizes most grouping analytics in existence today, (b) separates the relational concept from the analysis (grouping) concept – two different things according to our view, (c) can lead to rich optimization frameworks, and (d) provides a formal (and semi-declarative) base for MapReduce – which, in essence, depicts a similar idea. For example, given a relation  $B$  of all 2009's dates, the associated set (instance)  $\{S_b = \{s \text{ in Sales, such that Sales.date} \leq b \text{ and Sales.year} = 2009\}, b \in B\}$  could be used to compute the daily cumulative sales of 2009. Note that an associated set instance is just a collection of multisets. Although aggregation is a separate process, significant optimization can take place for the built-in aggregate functions.

An ASSET query consists of the computation of one or more associated sets, recursively defined: starting from a base table  $B_0$ , associated set  $(i+1)$  uses as its base table the base table of associated set  $i$  extended by its aggregates. We argue that a significant class of data analysis queries (especially continuous) can be easily represented and efficiently evaluated through this formalism. One can think this repeated, successive nature of query formulation as a spreadsheet document, where a column (cell) uses previously defined columns (cells) for its specification. In our view, ASSET queries can be useful in:

- (a) *incorporating heterogeneous data sources*: the data source of an associated set can be anything with a relational iterator defined over it – different database vendors, flat files, even the output of the query defined so far.
- (b) *distributed OLAP computation*: if the data source of an associated set is distributed to more than one nodes, the subset formation process can be easily distributed to these nodes – that does not mean that the associated set is materialized.
- (c) *performance*: since we can not apply traditional relational optimization over the data sources, we shift/replicate the optimization process (indexing, decorrelation, specialized join algorithms, distributed computation) to the data structure representing the ASSET query answer – which can always be made memory resident.

While a complete theoretical framework is essential – particularly for reasons described in Section 6 – we have so far focused on building tools and systems to support ASSET queries. We review these prototypes by providing motivating examples, architectural design and optimization techniques implemented. An ASSET query can be formulated via an extended SQL syntax or a spreadsheet-like GUI, called DataMingler. Its intermediate representation (XML-based) is fed to the COSTES system (Section 4), if the query involves continuous sources, or the ASSET Query Engine (Section 5), if the query involves persistent sources (Fig. 1) Although these two systems have been developed in parallel and kept distinct, we are investigating a common query processor for ASSET queries.

## 2 Grouping Analysis: A Retrospective

The need of complex data analysis involving aggregation over groups of data became apparent since the conception of database management systems. While the `group by`

clause was sufficient at the beginning, the dawn of new applications in the last ten years, such as web analysis, social networks, stream monitoring and others, necessitated advanced grouping constructs (cubes, grouping variables, windows) and novel programming paradigms such as MapReduce.

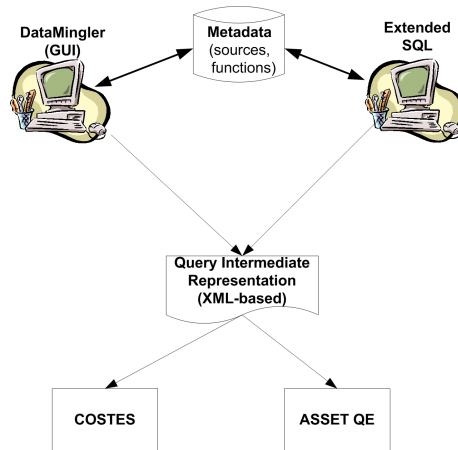


Fig. 1. Expressing and processing ASSET queries

## 2.1 Group By

Grouping was the first approach in database theory to support data analysis: the relation is *partitioned* based on one or more attributes and column-based aggregates are computed over each partition. Modeled as a relational operator (e.g. [1]), with multiple implementation algorithms (e.g. [2]), incorporated in query optimization (e.g. [3], [4]) and with a simple SQL syntax, it became an essential part of any DBMS. For complex analytics, users have to rely on multiple view definitions or nested queries. Usually, most commercial systems' performance break in queries representing trends, correlations or hierarchical aggregation. According to our view, group by, by following an ad-hoc approach to its definition, failed to separate the subset formation and the aggregation phase as two distinct processes. However, had it been modeled in a more generic way at that early point, it would have lost its simplicity and thus, its wide acceptance.

## 2.2 Cubes

With the rise of data warehousing and OLAP [5] came the need of multi-dimensional analysis, i.e. aggregations over multiple combinations of group by attributes. While traditional group by could be used to express and evaluate multidimensional analysis, there were significant linguistic and implementation benefits in introducing a new grouping construct, called cube [6], which computes an aggregate over all possible subsets of an attribute set. The *cube by* clause, an SQL syntactic sugaring extension,

made it easier for users and allowed the optimizer to use efficient evaluation algorithms [7], [8] to compute the cube – mainly by overlapping computation. While a major breakthrough, it lacked – according to our opinion – the aspect of separating the base values definition and the subset formation process as two distinct phases. For example, one may want to provide an ad-hoc set of group by attribute combinations and not the entire powerset, or compute multiple aggregations constrained by different  $\theta$  conditions for the same group by attributes (e.g. [9])

### 2.3 Grouping Variables and the MD-Join

Grouping variables [10] is the predecessor of associated sets and was an attempt to define multiple subsets, possibly overlapping and correlated, and perform aggregation over them. The work on grouping variables has influenced at least one commercial system and the standardization of ANSI SQL – OLAP amendment. Implementations of it have been studied in the context of telecom applications, medical and bio-informatics, finance and others [11]. Its deficiency was the insistence on the group by concept: formed subsets were tied to a group (subsets of a group) instead to a value. [12], [13] identified this and allowed subsets formation over the entire relation, but still the proposed SQL syntax was based on the group by syntax. The MD-Join operator [14], [15] modeled grouping variables in relational algebra and was one of the first aggregate-join operators: there was a set of base values  $B$  (a table, the base-values table), a detailed relation  $R$  and a condition  $\theta$  that was selecting subsets of  $R$ , assigned to a value of  $B$ . Aggregation of these subsets was coming next. The main claim of [14] was that we should separate the base-values definition phase and the aggregation phase for expressivity and efficiency reasons. Using MD-Join, many well-known algorithms [8] could be modeled through algebraic transformation rules proposed in that paper. The whole idea resembled an outer join [16]. It seems that outer joins are important in modern data analysis and spreadsheets are quite appropriate to express multiple outer-joins, placed one next to the other.

### 2.4 Windows

SQL/OLAP Amendment introduced certain new features in SQL language to support on-line analytical processing. One of the significant extensions is the ability to define windows over rows. A window enables users to determine the set of rows over which calculations can be performed with respect to the current selected row. In detail, in a window clause declaration we can define: the attribute list used for partitioning, the ordering of rows within partitions and an aggregation group. The aggregation group specifies which rows of each partition, with respect to the current row under examination, should participate in the evaluation of declared aggregate functions. While the idea of an associated set is present in the window construct, the contents of each set is predefined and thus, limited.

The window construct has been also used in data stream processing [17]. Due to the infinite nature of data streams we are interested only in a subset of the complete stream. Windows can limit the unbounded size of a data stream by defining a logical

or physical window over stream data. Stream data flow in and out of the defined window and functions computed continually over window transient data.

## 2.5 MapReduce

MapReduce [18] is one of the most active research areas during the last few years. It's a programming paradigm, consisting of two phases, modeled as functions: the mapping phase, where a set of values is derived, each associated with a list of values, and the reduce phase, where each list is reduced by some ad hoc aggregation method. The approach is similar to MD-Join and grouping variables. In fact, the distributed processing described of [18] is given relationally in [14] – but in the context of relations and not arbitrary data sources.

The claim of MapReduce is that with appropriate configuration of the Map and Reduce functions, a large number of computational tasks can be easily represented and efficiently executed. While this approach offers significant procedural flexibility over declarative approaches and employs a simple computational model, it lacks the optimizability and ease of use of modern database systems [19],[20]. While we completely agree with these claims, the ability to loop over the values of a domain and define an (associated) data set for each value is quite appealing both in terms of representation and evaluation. The goal is to balance the trade-offs between declarative optimizability and procedural flexibility in a database-proper way, such as in [21] and [22]. The trick in our case is to “restrict” the Map function to have a declarative nature while the Reduce function can be anything.

## 3 Associated Sets (ASSET) Queries

Based on the discussion of Section 2, a *data analysis* consists of three processes: (a) the definition of a set of base values  $B$ , (b) the specification of a collection  $A$  of subsets of a data source, one for each value of  $B$ , and (c) the aggregation method to apply on each member of  $A$ . If  $B$  is a relation, then we have something similar to nested relational models [23] or set-valued attributes [24].

### 3.1 Definitions

We want to define a collection of data sets, where each member of the collection is annotated by a tuple of a relation  $B$ , i.e.  $\{S_b, b \in B\}$ . These sets are populated by a data source  $S$ . Formally,

**Definition 3.1.** Given a relation  $B$  and a data source  $S$  with relational schema  $\mathcal{S}$ , then any set  $A = \{S_b: b \in B, S_b \text{ a multiset of tuples having schema } \mathcal{S}\}$  is called an associated set (instance) with schema  $(B, \mathcal{S})$ .  $B$  is called the base relation of  $A$  and  $S$  the data source of  $A$ .  $\square$

This definition is appropriate for both transient and persistent data sources. However, if the data source is persistent, then we may use a condition  $\theta$  involving attributes of  $B$  and  $S$  to define a specific associated set instance.

**Definition 3.2** Given a relation  $B$  and a *finite* data source  $S$  with relational schema  $S$  and a condition  $\theta$  involving attributes of  $B$  and  $S$ , then the associated set  $A = \{S_b: b \in B, S_b \text{ a subset of } S \text{ such that } \theta(b,s) \text{ is true } \forall s \in S_b\}$  is called the associated set (instance) with respect to  $S$  and  $\theta$ .  $B$  is called the base relation of  $A$ ,  $S$  the data source of  $A$  and  $\theta$  the defining condition of  $A$ .  $\square$

We can use a condition  $\theta$  over infinite data sources (and we do in COSTES), however the members of an associated set  $A$  are not monotonically increasing during evaluation, as in finite data sources.

Definition 3.1 cleanly separates the base values that we use to form the subsets, subset formation and subsequent aggregation.

### 3.2 SQL Syntax

We follow the formalism of grouping variables in [13]. The idea is quite simple: we want a syntax that allows the addition of “extra” columns to the resulting table of an SQL query – similar to an outer-join operation. We propose an “extended by” clause to declare the associated sets and their respective data sources and a “such that” clause to provide their defining conditions. These clauses immediately follow a `<select..from..where..group by>` query. The proposed syntax is as follows:

```
select A from R where  $\theta$  group by A'
extended by  $A_1(S_1), A_2(S_2), \dots, A_n(S_n)$ 
such that  $\theta_1, \theta_2, \dots, \theta_n$ 
```

The selection list  $A$  may contain aggregate functions defined over the associated sets  $A_1, A_2, \dots, A_n$ . The answer of the `<select..from..where..group by>` SQL query serves as the base-values table. Condition  $\theta_i$  involves attributes of the base-values table, constants and aggregates of associated sets  $A_1, \dots, A_{i-1}, i=1, \dots, n$ .

### 3.3 DataMingler: A Spreadsheet-Like GUI

While an SQL extension is a mandate to our SQL-centric universe, spreadsheet-like query tools have been praised for their simplicity and flexibility [25], [26], [27]. We consider spreadsheet-like formulations particularly useful to express ASSET queries. An associated set uses the already defined spreadsheet as its base table, its defining condition involves columns of the spreadsheet and the associated set’s data source – resembling a spreadsheet formula – and its aggregates become new columns of the spreadsheet. By doing this recursively, into a single table, users can build very powerful reports, sometimes closer to programs rather than traditional database queries (e.g. we can use ASSET queries for data reconciliation in financial applications.) For continuous queries, given that the data source of an associated set can be the spreadsheet itself, dependencies between associated sets dictate the order of update of the ASSET structure.

We have developed a spreadsheet-like GUI, called DataMingler, to manage data sources, user-defined aggregate functions and ASSET queries. It has been

implemented in C++ code using the Qt4 C++ library from Trolltech (platform independent).

**Data Source Management:** An ASSET query may use heterogeneous and possibly multi-partitioned data sources. These sources may refer to local or remote databases, data streams or flat files and must firstly be appropriately defined through DataMingler. Each description consists of the source's schema and a number of attributes specific to the type of the source (e.g. delimiter and location for flat files; IP, port, username and password for databases, etc.) All data sources are stored in an XML-based specification file. Currently, DataMingler support databases (PostgreSQL, MySQL), flat files and socket-based streams. All data sources may consist of multiple partitions, not necessarily of the same schema – only common attributes appear in query formulation. A partition in the case of databases/flat files/data stream is just another table/file/stream source, located locally or remotely. As a result, a data source may consist of multiple tables/files/streams distributed to several processing nodes.

**Aggregate Functions:** The goal is to describe the signature of a C++ function into an XML-based dictionary, so some type-checking and user-guidance can take place. The user specifies the input parameters and their types and the type of the return value. S/he also specifies a “gluing” function, in the case of distributed computation of an associated set (e.g. “sum” is the gluing function for “count”). Aggregate functions can be either distributive or algebraic (holistic computations can be achieved through aggregate functions returning the entire or part of the associated set and the use of “null” associated sets, described later). In the case of algebraic aggregate functions, the user must specify the involved distributive functions, the correspondence between the parameters and the finalization code (in C++).

**ASSET Queries:** Users specify ASSET Queries through DataMingler in a spreadsheet-like manner, column by column. The user initially specifies a base-values table that can be an SQL query over one of the database sources, the contents of a flat file source or manually inserted schema and values. Thus, the first columns of the spreadsheet correspond to the base-values table attributes. The spreadsheet document is then extended with columns representing associated sets, one at a time. The user specifies the name, source, defining condition and aggregate functions of the associated set. The data source can be (a) one of the existing data sources described earlier through DataMingler, (b) of type “this”, in which case the so-far defined spreadsheet table serves as the data source to the associated set, and (c) of type “null”, in which case the user specifies an expression involving aggregates of previously defined columns – similar to a spreadsheet formula involving only same-row cells. Associated sets may be correlated, since aggregations performed over one associated set may be used by another. This might occur during specification of the latter's defining condition, its functions' parameters or its computation formula in case of “null” sets.

## 4 ASSET Queries and Data Streams (COSTES)

We have used associated sets to express complex continuous queries for financial data analysis [28] and RFID data processing [29]. In the following sections we present

some financial data analysis examples to show the notion of ASSET continuous queries. More examples can be found in [28], [29].

#### 4.1 Financial Application Motivating Examples

Real time financial data analysis provides the means to organizations to make faster trading transactions and monitor transaction's performance. As an example, consider a financial application with the following schema:

```
Stocks(stockID, categoryID, description)
```

and the presence of data streams reporting continually stock ticks and stock's trading volume:

```
Prices(stockID, timestamp, price)
Volumes(stockID, volume, timestamp)
```

Financial analysts may register the following continuous queries to monitor stock activity:

- Q1. Monitor for each stock the minimum, maximum and average price that has been seen so far. With this query we can detect severe fluctuations of a stock's performance at real time.
- Q2. We want for each stock to continuously know when its average price of the last 10 reported prices is greater than its running average price. In that case, a "True" value should appear next to the stock id, otherwise a "False" is displayed. This query can be used to alert analysts for "hot" periods of a stock.
- Q3. In many occasions it is useful to express correlated aggregation [10], [30] in the context of data streams, i.e. use a continuously aggregated value to constraint a subset of stream data. We may be interested in monitoring the running total volume of each stock, but summation should take place only when the average price of the last 10 reported prices is greater than the running average price of the stock. Then, we want to contrast this with the (regular) running total volume. This query can show periods of time of increased volume traffic.

Fig. 2 shows ASSET queries formulation and instances of the results. In Q1, we define for each stockID an ordered sequence (X associated set) to keep the reported prices that have been seen so far and then compute the min, max and average price over it. This is a running window in stream literature, however the idea is to declaratively define the set to keep the stream tuples and let the optimizer choose the most appropriate data structure for implementation. Similarly in Q2, we define two associated sets using the stock IDs as the base table, X and Y, X to store the reported prices since the registration of the query and Y to stores the last 10 prices, using the `size()` function in the defining condition of Y. Note that Q1 and Q2 do not use the `size()` function for associated set X. In this case we assume that we have an infinite

associated set per stockID and compute the maximum and average over it. Finally in Q3 we take a similar approach as in Q1 but we constrain the Z associated set, using aggregates of associated sets X and Y. If the average price of associated set Y is greater than the average price of associated set X, we append Volume's tuple to Z. In all examples, the evaluation approach is similar, for each stock we keep the price/volume values and compute the specified aggregate functions. For example in Q2 the evaluation algorithm is presented below:

```

for each stockID s in Stocks {
  Xs = {v in Prices: v.stockID == s};
  Ys = {v in Prices: v.stockID == s and Y.size(10)};
  compute (Y.avg(price)>X.avg(price));
}

```

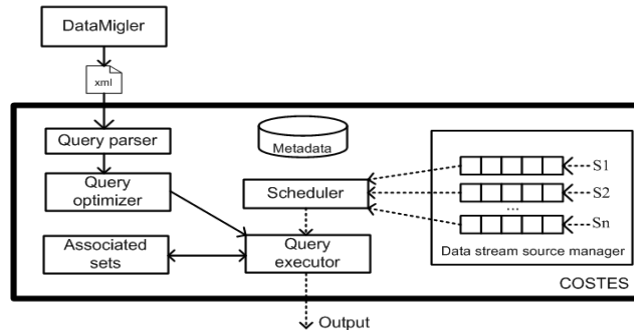
The initial idea of COSTES was to express spreadsheet-like reports, where previously defined columns may constrain later columns. These columns are aggregates of associated sets. We claimed in [29] that using this framework, one can express many practical continuous queries, which have been given little attention so far. In addition, there are numerous implementation choices for the optimizer: most appropriate data structures to represent the associated sets; indexing methods over these associated sets; overlapping associated sets. Finally, the addition/deletion of associated sets should take place during runtime. These ideas have been implemented in COSTES.

<p>Q1:  select stockID, X.min(price) as min_price,  X.max(price) as max_price,  X.avg(price) as avg_price  from Stocks  extended by X(Prices)  such that X.stockID = stockID</p>	<table border="1"> <thead> <tr> <th>stockID</th> <th>min_price</th> <th>max_price</th> <th>avg_price</th> </tr> </thead> <tbody> <tr> <td>MSFT</td> <td>29.12</td> <td>29.31</td> <td>29.15</td> </tr> <tr> <td>ORCL</td> <td>19.12</td> <td>19.19</td> <td>19.17</td> </tr> <tr> <td>BAC</td> <td>54.48</td> <td>54.81</td> <td>54.67</td> </tr> <tr> <td>...</td> <td>...</td> <td>...</td> <td>...</td> </tr> <tr> <td>GM</td> <td>35.35</td> <td>35.87</td> <td>35.54</td> </tr> </tbody> </table>	stockID	min_price	max_price	avg_price	MSFT	29.12	29.31	29.15	ORCL	19.12	19.19	19.17	BAC	54.48	54.81	54.67	...	...	...	...	GM	35.35	35.87	35.54
stockID	min_price	max_price	avg_price																						
MSFT	29.12	29.31	29.15																						
ORCL	19.12	19.19	19.17																						
BAC	54.48	54.81	54.67																						
...	...	...	...																						
GM	35.35	35.87	35.54																						
<p>Q2:  select stockID, (Y.avg(price)&gt;X.avg(price)) as Flag  from Stocks  extended by X(Prices), Y(Prices)  such that X.stockID = stockID,  Y.stockID = stockID and Y.size() = 10</p>	<table border="1"> <thead> <tr> <th>stockID</th> <th>Flag</th> </tr> </thead> <tbody> <tr> <td>MSFT</td> <td>True</td> </tr> <tr> <td>ORCL</td> <td>False</td> </tr> <tr> <td>BAC</td> <td>False</td> </tr> <tr> <td>...</td> <td>...</td> </tr> <tr> <td>GM</td> <td>True</td> </tr> </tbody> </table>	stockID	Flag	MSFT	True	ORCL	False	BAC	False	...	...	GM	True												
stockID	Flag																								
MSFT	True																								
ORCL	False																								
BAC	False																								
...	...																								
GM	True																								
<p>Q3:  select stockID, Z.sum(volume) as sum_volume_10,  W.sum(volume) as sum_volume  from Stocks  extended by X(Prices), Y(Prices), Z(Volume), W(Volume)  such that X.stockID = stockID,  Y.stockID = stockID and Y.size() = 10,  Z.stockID = stockID and Y.avg(price)&gt;X.avg(price),  W.stockID = stockID</p>	<table border="1"> <thead> <tr> <th>stockID</th> <th>sum_volume_10</th> <th>sum_volume</th> </tr> </thead> <tbody> <tr> <td>MSFT</td> <td>29.17</td> <td>29.28</td> </tr> <tr> <td>ORCL</td> <td>19.14</td> <td>19.17</td> </tr> <tr> <td>BAC</td> <td>54.68</td> <td>54.68</td> </tr> <tr> <td>...</td> <td>...</td> <td>...</td> </tr> <tr> <td>GM</td> <td>35.42</td> <td>35.76</td> </tr> </tbody> </table>	stockID	sum_volume_10	sum_volume	MSFT	29.17	29.28	ORCL	19.14	19.17	BAC	54.68	54.68	...	...	...	GM	35.42	35.76						
stockID	sum_volume_10	sum_volume																							
MSFT	29.17	29.28																							
ORCL	19.14	19.17																							
BAC	54.68	54.68																							
...	...	...																							
GM	35.42	35.76																							

Fig. 2. ASSET queries and instances of results of queries Q1, Q2 and Q3.

## 4.2 COSTES: Continuous Spreadsheet-like Computations

We have developed a system prototype called COSTES [29] which supports ASSET continuous queries. Fig. 3 depicts COSTES architecture:



**Fig. 3.** COSTES architecture

DataMigler provides a GUI interface to declare ASSET continuous queries. DataMigler generates an XML file, which contains an intermediate representation of the query. Query parser validates query's syntax and query optimizer analyzes the query for possible optimizations. Our optimizer provides the following optimizations:

- Build appropriate indexes according to *such that* predicate to quickly locate rows of the base table and avoid a full scan. For example, in all discussed queries we can build a hash index on Stocks.stockID.
- Parse the *select* and *such that* clauses to keep in corresponding associated sets only the attributes needed for the evaluation of aggregate functions and avoid holding all the attributes of stream tuples.

Once the base relation has been computed and loaded, the associated set structures are initialized and linked to the base relation. Associated sets manager module is responsible for initializing and maintaining associated sets structures. Scheduler retrieves tuples from input queues (one for each data source) and forwards them to query executor for processing. Input queues are handled by Data Stream Source Manager, which supports concurrent data retrieval from various data sources i.e. flat files, databases, network sockets streams and XML sources. Finally, metadata catalog contains information such as data source names and types, schemas, etc and is used by other modules during query initialization.

## 5 ASSET Queries and Persistent Data Sources (ASSET QE)

We have used ASSET Queries in the context of distributed data warehouses<sup>1</sup>. We claim that the structure of an ASSET query is useful not only in expressing a practical class of OLAP queries, but also in developing an efficient optimization framework for distributed settings.

<sup>1</sup> Part of this research has been done while the first author was visiting AsterData Systems.

## 5.1 Social Networks: A Motivating Example

Assume a social network with a streaming service, where users can post their own videos and see others', similar to YouTube, MySpace or Google Videos. A (part of) the schema design is shown below:

```
VideoPageViews (userid, sessionid, videoid, timespent,...)
Users (userid, type, age, country, financial, ...)
Videos (videoid, categ, videotype, ownerid, duration, ...)
```

In many occasions it is useful to know the dominant (most frequent) category of videos each user over 25 watches. An SQL formulation would be the following:

```
create view UC (userid, categ, cnt) as
select u.userid, s.categ, count(*)
from VideoPageViews v, Users u, Videos s
where v.userid=u.userid and s.videoid=v.videoid and u.age>25
group by u.userid, s.categ;

select UC.userid, UC.categ
from (select userid, max(cnt) as max_cnt
      from UC
      group by userid) as G, UC
where UC.userid=G.userid and UC.cnt=G.max_cnt;
```

While this representation is not particularly difficult, an alternative, semi-procedural, set-oriented approach, seems more intuitive to some analysts: for each user, form the set of all videos she has seen and keep the videoids; match these videoids with the corresponding category; find the most frequent element of this set:

```
for each user u in Users over 25{
  Xu = {v in VideoPageViews: v.userid == u};
  Yu = {v in Videos: v.videoid in Xu.all(videoid)};
  compute mostOften(Yu.all(categ)); }
```

In our extended SQL, this query can be expressed as:

```
select userid, mostOften(Y.categ)
from Users
where age>25
extended by X(VideoPageViews), Y(Videos)
such that X.userid=userid,
         Y.videoid in X.all(videoid)
```

Note that associated sets can have set-valued aggregates. An efficient evaluation of this query would involve the following steps:

- (a) build a hash index on `userid` on `Users`
- (b) compute associated set `X`: scan `VideoPageViews` and match with the corresponding `userid`. Since `X`'s aggregate (`X.all(videoid)`) will only be used conjunctively in `Y`'s defining condition to test membership, it can be kept as an inverted list (videoids pointing to row numbers) instead as a data set.
- (c) compute associated set `Y`: scan `Videos` and use the inverted list to match `Videos` tuples with row numbers. The `mostOften` aggregate function is implemented by keeping `(categ, counter)` pairs hash-indexed on `categ`.

If VideoPageViews is distributed to several processing nodes, the computation of associated set X is also simple: send all userids to each node, compute the partial associated set X and “glue” (union) the partial results at the coordinator (a special node). In other words, we consider that while ASSET queries still retain a relational flavor, a query processor can reason easier on distributed settings. The goal of an ASSET query engine is to implement such optimizations and conclude to an efficient execution plan as the one described above. Given current main-memory sizes and disk configurations, such evaluation plans are feasible.

Fig. 4 shows ASSET QE performance on this query, varying VideoPageViews size from 100M to 600M records (15GB to 90GB) – all in one partition. We assumed 10M users and 10K videos. Using standard SQL on PostgreSQL DBMS did not return any results for 200M records for at least 2 hours. All experiments performed on a Linux Dell machine with a Quad Core Intel Xeon CPU @ 3.00GHz having 12 disks, 300GB each at 15K rpm, RAID5 configuration and 32GB of main memory.

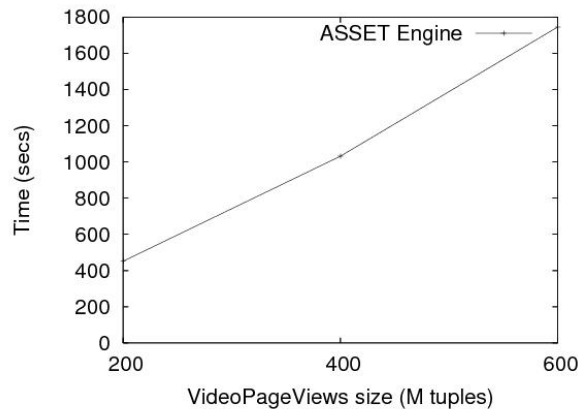


Fig. 4. ASSET QE performance

## 5.2 ASSET Query Engine (QE)

Once an ASSET query involving persistent data sources has been formulated and represented as an XML-based document, it is passed to the ASSET QE for optimization, code generation (C++) and execution. The goal is the efficient computation of the involved associated sets and their aggregates. ASSET QE performs the following two major steps:

- (a) since there may be dependencies between associated sets (an associated set’s aggregate may be used in the defining condition or aggregates of another), we must first assign associated sets to “processing rounds”, with no intra-round dependencies.
- (b) each processing round involves several data sources, possibly partitioned to several processing nodes. We must derive the per-partition associated set list, and generate an efficient program that computes the associated sets with respect to the partition. This program can execute at the coordinator

(partition's data transferred over the network) or at the partition's host (aggregates are serialized back to the coordinator).

The architecture of our system is shown in Fig. 5.

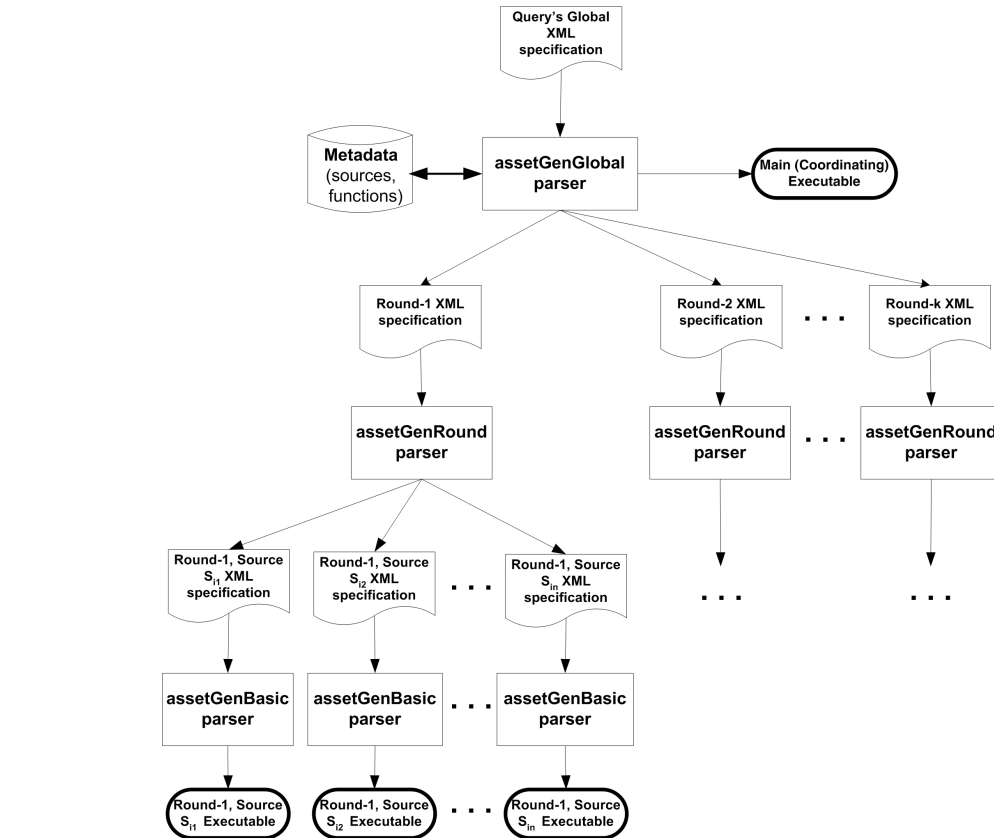


Fig. 5. Parsing and code generation of an ASSET query

**assetGenGlobal:** This is the top-level parser of the ASSET QE. It gets the XML-based specification of an ASSET query and generates (a) the round-related XML specifications of the query and (b) the main (coordinating) C++ program for the query. Each round-related specification contains the data sources' description of the round and the associated sets that will be computed. Note that from this point on, each partition of a data source becomes a distinct, individual data source. The query's main C++ program, instantiates and populates all the necessary data structures, creates all the local indexes and decorrelation lists over the ASSET structure and coordinates all the *basic computational threads* (discussed later) executing locally or remotely. In the latter case, it sends parts of the ASSET structure to the appropriate nodes and receives back (and glues together) the computed column(s).

**assetGenRound:** This is the round-level parser: it groups the associated sets of the round by source and generates an XML-based specification file for each source.

Recall that with the term “source” we mean partitions of the original data sources. It determines whether the computation over the source will execute locally or remotely, deduces the indexes and decorrelation lists over the base-values table and resolves the minimal base-values table that has to be sent to the remote node (in case of remote computation.) Currently supported indexes are hash maps, binary trees and inverted lists, deduced by the defining condition of the associated sets.

**assetGenBasic:** This is the source-level parser that gets a source-specific XML-based specification file and generates an efficient C++ program (a “basic computational thread”) to scan the data source and compute the associated sets related to that source. The basic computational thread communicates with the main program to receive the round-specific base table (only the required columns), builds indexes over and decorrelates the base table, computes the associated sets and serializes the result back to the coordinating program (if executing remotely). The engine also decides to decorrelate the base table on a single attribute with respect to an associated set (i.e. we may have different decorrelation lists for different associated sets), if the associated set is using a hash index on that attribute and its estimated cardinality is low (this can be measured while receiving the base table).

Once all the basic computational threads have been generated, then the whole process is driven by the query’s main C++ program. We currently assume that the entire ASSET structure fits in main memory – which is not unrealistic for a large class of ASSET queries and today’s memory sizes. However, since the entire code generation assumes boundary limits of the ASSET structure, we can easily specify the computation of an ASSET query in horizontal chunks, but it has to be done manually, by altering the query’s main C++ program. Fig. 6 depicts the (simple) execution plan of an ASSET query.

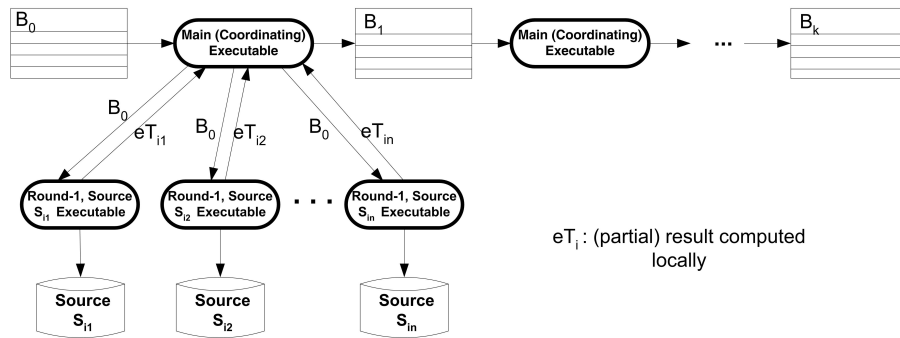


Fig. 6. Simple evaluation plan of an ASSET query

## 6 Conclusions and Future Work

In this paper we presented the concept of associated set and briefly presented two prototypes using associated sets, one in data streams and one in persistent data sources. We believe that ASSET queries show promise both linguistically and

computationally. Little has been done in terms of theoretical work though, since the focus was rapid prototyping. It seems challenging however, because it is a restricted form of second order predicate logic. In general, given a relation  $R$ , the powerset  $P$  of  $R$ ,  $\mathcal{P} = \text{Pow}(\mathcal{R})$ , contains all possible subsets of  $R$ . Group-bys, grouping variables, windows, MapReduce, etc. are just ad hoc (and efficient) constructs to denote specific subsets of  $\mathcal{P}$ . What would be a generic language  $\mathcal{L}$  to specify subsets of  $\mathcal{P}$ ? For example, how one can express the following query: “find the average income for all sets of four users who were born before 1940”? What would be an efficient, yet flexible sub-language of  $\mathcal{L}$ ? What would be its expressive power? We would like to investigate whether associated sets can form the basis for such research.

## References

1. Elmasri, R., Navathe, S.B.: Fundamentals of Database Systems. Addison-Wesley, (1994)
2. Graefe, G.: Query Evaluation Techniques for Large Databases. In: ACM Computing Surveys, 25, 73—170 (1993)
3. Chaudhuri, S., Shim, K.: Including Group-By in Query Optimization. In: 20th International Conference on Very Large Data Bases, pp. 354--366. Morgan Kaufmann, (1994)
4. Yan, W.P., Larson, P.: Eager Aggregation and Lazy Aggregation. In: 21st International Conference on Very Large Data Bases, pp. 345--357. Morgan Kaufmann, (1995)
5. Chaudhuri, S., Dayal, U.: An Overview of Data Warehousing and OLAP Technology. SIGMOD Record, 26, 65--74 (1997)
6. Gray J., Bosworth A., Layman A., Pirahesh H.: Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Total. In: 12th International Conference on Data Engineering, pp. 152--159. IEEE Computer Society, (1996)
7. Agarwal, S., Agrawal, R., Deshpande, P., Gupta, A., Naughton, J.F., Ramakrishnan, R., Sarawagi, S.: On the Computation of Multidimensional Aggregates. In: 22nd International Conference on Very Large Data Bases, pp. 506--521. Morgan Kaufmann (1996)
8. Ross, K.A., Srivastava, D.: Fast Computation of Sparse Datacubes. In: 23rd International Conference on Very Large Data Bases, pp. 116--125. Morgan Kaufmann (1997)
9. Ross, K.A., Srivastava, D., Chatziantoniou, D.: Complex Aggregation at Multiple Granularities. In: Schek, H., Saltor, F., Ramos, I., Alonso, G. (eds.) EDBT 1998. LNCS, vol. 1377, pp. 263--277. Springer, (1998)
10. Chatziantoniou, D., Ross, K.A.: Querying Multiple Features of Groups in Relational Databases. In: 22nd International Conference on Very Large Data Bases, pp. 295--306. Morgan Kaufmann, (1996)
11. Chatziantoniou, D.: Using grouping variables to express complex decision support queries. Data Knowl. Eng. 61, 114--136 (2007)
12. Chatziantoniou, D.: Evaluation of Ad Hoc OLAP: In-Place Computation. In: 11th International Conference on Scientific and Statistical Database Management, pp.34—43. IEEE Computer Society, (1999)
13. Chatziantoniou D.: The PanQ Tool and EMF SQL for Complex Data Management. In: 5th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp.420—424. ACM (1999)
14. Chatziantoniou, D., Akinde, M.O., Johnson, T., Kim, S.: The MD-join: An Operator for Complex OLAP. In: 17th International Conference on Data Engineering, pp. 524—533. IEEE Computer Society, (2001)
15. Akinde, M.O., Böhlen, M.H., Johnson, T., Lakshmanan, L.V.S., Srivastava, D.: Efficient OLAP Query Processing in Distributed Data Warehouses. In: Jensen, C.S., Jeffery, K.G.,

- Pokorny, J., Saltenis, S., Bertino, E., Bohm, K., Jarke, M. (eds.) EDBT 2002. LNCS, vol. 2287, pp. 336--353. Springer, (2002)
16. Steenhagen, H.J., Apers, P.M.G., Blanken, H.M.: Optimization of Nested Queries in a Complex Object Model. In: Jarke, M., Bubenko, J.A., Jeffery, K.G. (eds.) EDBT 94. LNCS, vol. 779, pp. 337--350. Springer, (1994)
  17. Babcock, B., Babu, S., Datar, M., Motwani, R., Widom, J.: Models and Issues in Data Stream Systems. In: 21st ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, pp. 1—16. ACM (2002)
  18. Dean, J., Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters. In: 6th Symposium on Operating System Design and Implementation, pp. 137—150. USENIX Association, (2004)
  19. DeWitt, D.J., Stonebraker, M.: MapReduce: A major step backwards. The Database Column, <http://www.databasecolumn.com/2008/01/mapreduce-a-major-step-back.html>.
  20. Pavlo, A., et al.: A Comparison of Approaches to Large-Scale Data Analysis. In ACM SIGMOD Conference, pp. 165--178. ACM (2009)
  21. Olston, C., Reed, B., Srivastava, U., Kumar, R., Tomkins, A: Pig latin: a not-so-foreign Language for Data Processing. In: SIGMOD Conference, 2008, pp. 1099-1110. ACM (2008)
  22. Abouzeid, A., et al.: HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads. In VLDB Conference, 2009 (to appear)
  23. Roth, M.A., Korth, H.F., Silberschatz, A.: Extended Algebra and Calculus for Nested Relational Databases. ACM Trans. Database Syst. 13, 389--417 (1988)
  24. Mamoulis, N.: Efficient Processing of Joins on Set-valued Attributes. In: ACM SIGMOD International Conference on Management of Data, pp. 157--168. ACM (2003)
  25. Winslett, M.: Interview with Jim Gray. SIGMOD Record 32, 53--61 (2003)
  26. Witkowski, A., Bellamkonda, S., Bozkaya, T., Dorman, G., Folkert, N., Gupta, A., Sheng, L., Subramanian, S.: Spreadsheets in RDBMS for OLAP. In: ACM SIGMOD International Conference on Management of Data, pp. 52--63. ACM (2003)
  27. Liu, B., H.V. Jagadish, H.V.: A Spreadsheet Algebra for a Direct Data Manipulation Query Interface. In: 25th International Conference on Data Engineering, pp. 417--428. IEEE (2009)
  28. Chatziantoniou, D., Sotiropoulos Y.: Stream Variables: A Quick but not Dirty SQL Extension for Continuous Queries. In: 23rd International Conference on Data Engineering Workshops, pp. 19--28. IEEE Computer Society, (2007)
  29. Chatziantoniou, D., Sotiropoulos Y.: COSTES: Continuous spreadsheet-like computations. In: 24th International Conference on Data Engineering Workshops, pp. 82—87. IEEE Computer Society, (2008)
  30. Gehrke, J., Korn, F., Srivastava, D.: On Computing Correlated Aggregates Over Continual Data Streams. In: ACM SIGMOD International Conference on Management of Data, pp. 13--24. (2001)