ASSET Queries: A Declarative Alternative to MapReduce

Damianos Chatziantoniou*
Department of Management Science and Technology,
Athens University of Economics and Business (AUEB)
damianos@aueb.gr

Elias Tzortzakakis
Institute of Computer Science
Foundation for Research and Technology (FORTH)
tzortzak@ics.forth.gr

ABSTRACT
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 or flat files, or can be generated in real-time in a continuous manner. An associated set is a collection of data sets, annotated by the values of a domain D. These data sets are populated using a data source according to a condition θ and the annotated value. An ASsociated SET (ASSET) query consists of repeated, successive, interrelated definitions of associated sets, put together in a column-wise fashion, resembling a spreadsheet document. We present DataMingler, a powerful GUI to express and manage ASSET queries, data sources and aggregate functions and the ASSET Query Engine (QE) to efficiently evaluate ASSET queries. We argue that ASSET queries: a) constitute a useful class of OLAP queries, b) are suitable for distributed processing settings, and c) extend the MapReduce paradigm in a declarative way.

Categories and Subject Descriptors
H.2.3 [**Database Management**]: Languages—*Query Languages*

General Terms
Design, Management, Languages.

Keywords
MapReduce , Spreadsheets, ASSET Queries, DataMingler.

1. INTRODUCTION
Business needs of modern applications require advanced data analysis over voluminous data sets, usually partitioned across different disks or processing nodes, possibly in different formats (e.g. flat files and/or multiple-vendor databases). To accommodate the enormous processing requirements of these applications, novel hardware/database architectures have been proposed (e.g. [8],[17]) and programming paradigms have been developed (e.g. MapReduce [9].) In addition, several well-funded start-ups, such as AsterData, Greenplum, Netezza and Vertica now offer products for large-scale data analytics alongside with IBM, Oracle and Teradata.

The goals of this work are the following:

(a) Distributed computation of OLAP queries over very large data warehouses: a data warehouse may be distributed to several nodes for reliability, load-efficiency and cost-efficiency reasons. An interesting research question is what kind of OLAP questions can be efficiently evaluated over distributed data warehouses, and how much and which part of the computation one can "move" to the node hosting a partition.
(b) Simple query formulation: representation of a query should be as simple and intuitive as possible, yet amenable to appropriate distributed processing rewrites. MapReduce is an option, but it lacks declarative simplicity ([1],[10],[14]). In addition, there is a plethora of OLAP queries that could benefit from a MapReduce implementation, but cannot be expressed as MapReduce jobs (e.g. pivoting, hierarchical comparisons, complex comparisons, trends, correlated aggregation [7].)
(c) Heterogeneous data sources: data sources can be persistent or continuous, databases of different vendors, or flat files. In general, a data source can be anything that presents a relational interface to our system and has an iterator defined over it. While we briefly discuss ASSET queries in the context of data streams in this paper, a detailed presentation can be found in [4].

As a running example, consider a financial application with schema:

Clients(clientID, address, zip, income)
Stocks(stockID, categID, description)
Transactions(clientID, stockID, volume, timestamp, type, amount)

Typical data warehousing queries include:

Q1. For each stock, find certain demographics of its buyers (e.g. average income) and compare them to those of the stock's category.
Q2. Find the most frequent stock category a user buys or sells, for a predefined set of users.
Q3. For each stock, compare the demographics (e.g. average income) of its large-volume buyers versus its small-volume buyers.

* Part of this research was performed while the author was visiting Aster Data Systems Inc., Redwood Shores, CA.

Additional queries of this "style" can also be found in [6],[7]. They constitute a useful subclass of OLAP queries and share a pattern: for each tuple of a table, they compute a "define-subset, reduce-subset" sequence, where each "define-subset" phase of a step uses previously defined aggregates and/or the tuple's attributes. For example, consider query Q2. While SQL is the de facto option, one could formulate it in a stepwise, set-oriented fashion:

```
1:  for each user u {
2:    T={find the transactions of u};
3:    F={StockIDs of T's members};
4:    S={select the stocks with StockID in F};
5:    C= mostOften(S.categID);
6:    print (u, C);
7:  }
```

Line 2 defines a subset of transactions T; Line 3 defines a (set-valued) aggregate over T; Line 4 defines a subset of stocks; Line 5 defines an aggregate over this set. One could find such formulations easier than SQL, especially those with a procedural background. At the same time, such implementations, with the appropriate indexing, data structure selection and memory sizes can perform significantly better than current DBMS. Finally, if a data source is partitioned across nodes, parallel processing becomes "cleaner". Such queries resemble MapReduce jobs, but there are two notable differences: (a) there may be several reduce sets for the same map value, and (b) these reduce sets can be correlated.

Our approach is based on the concept of associated set (ASSET), which is just a collection of tuples associated to a value (not necessarily atomic). An ASSET query consists of one or more associated sets, put together in a specific way. It resembles a spreadsheet report, where earlier columns serve as the basis for later columns via formulas and the first few columns are somehow initialized to external values. In our framework, users initially define the base columns using a database or flat file and they then build incrementally the report by adding columns. Each cell of the new column represents a data set (the associated set), populated from a data source according to a condition θ, involving attributes of the source and aggregates of previously defined associated sets. We claim that this column-wise formulation of a query can be not only intuitive and flexible, but also efficient and robust in terms of associated sets' evaluation. Similar claims for spreadsheet-like query languages can be found in [3],[18].

In this paper, we present (a) the ASSET Query Engine (QE), which parses, optimizes and executes ASSET queries, and (b) DataMingler, a spreadsheet-like tool to express ASSET queries. In our view, ASSET queries can be useful in:

Novel Programming Paradigms: MapReduce [9] is one of the most active research issues over the last few years. The claim 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 [10]. While we completely agree with the claims of [10], the ability to loop over the values of a domain and define an (associated to that value) data set is quite appealing both in terms of representation and evaluation. It has been used, directly or indirectly, in parameterized query processing, in set-valued attribute proposals [13], in grouping constructs [6], relational join operators ([5],[16]) and elsewhere. The goal should be to balance the trade-offs between declarative optimizability and procedural flexibility in a database-proper way, such as in Hive [12] and Pig [14]. For example, Pig adds a semi-declarative layer over MapReduce, by proposing a language combining declarative and procedural features. In our approach, ASSET queries "restrict" the Map function to have a declarative nature while the Reduce function can be anything (using C++).

Performance: The answer to an ASSET query is represented by a – possibly nested – data structure, which can always be made memory resident (horizontally segmented, if necessary). This data structure can be indexed multiple times, using different methods; can be decorrelated, if parts of it contain the same data; can be computed locally or partitioned and sent to the nodes containing the data sources. The claim is that the representation of an ASSET query is appropriate to identify and employ the above-mentioned optimization techniques.

Integrating Heterogeneous Data Sources: By allowing a column's data source to be essentially an iterator over anything that presents a relational interface, we can integrate into the same report data sets from heterogeneous data sources.

SQL Extensions: Sticking to SQL for query formulation has traditionally been a "must do" for every proposal, given its popularity among database users. Fortunately, it seems that there is a simple and intuitive SQL extension that allows ASSET query formulation, following the syntactic paradigm of [7].

2. EVALUATION OF ASSET QUERIES

Let us re-visit query Q2. The basic idea is simple. First, for each client id (table B_0 in Figure 1), define the set of client's transactions $\mathcal{T}$, using as data source the Transactions table, and keep the stock ids of $\mathcal{T}$ as a set-valued aggregate function, called *all()*. These are the F_i sets in Figure 1. B_0 is then extended with a column containing F_i sets and is named B_1. Then, for each row *i* of B_1, define the set of stocks $\mathcal{S}$ with stockID in F_i, using as data source the Stocks table, and compute the

mostOften(categID) element of $\mathcal{S}$. The entire process of defining column-wise this query is depicted in Figure 1.

Figure 1: Representation of query Q2

Therefore the idea is to start from a set of base columns and add recursively new columns: for each row define a set of values using some condition θ (the associated set), compute one or more aggregates over these sets, extend the base schema with these aggregates and start over. The definition of the associated set is done declaratively, while the aggregate functions over the associated set is a C++ method with a simple- or vector-type return value. Each associated set's definition resembles a single MapReduce application with the mapping phase (initial values) already computed.

The claims are: (a) ASSET queries can *express* more complex OLAP queries than MapReduce, queries similar to [7], and (b) there are significant optimization opportunities in representing a query in such a way.

Let us revisit query Q2: in most distributed data warehousing configurations, Transactions table will be partitioned across several processing nodes, while Stocks table will be replicated at each node. One evaluation plan would be to distribute B_0 to all nodes, compute the partial S_i's, and ship them back to and concatenate them (union) at the coordinating node. The optimizer could deduce that S_1, S_2, etc. are used later only for membership testing and keep them as inverted lists (to the row ids) to facilitate the efficient computation of $\mathcal{S}$ sets. These optimizations are possible due to the representation of ASSET queries. One can argue that the size of B_1 may be large, but given current RAM sizes, it actually is very feasible and makes sense: maximize RAM's role as much as possible. The impact on performance by employing these optimizations techniques can be huge: from non-ending queries down to a few hundreds of seconds (Section 5).

Due to space limitations, we cannot define formally the concept of associated set. In short, given a set of base values V, an associated set A w.r.t. a data source S is a collection of *empty* data structures, able to hold S' elements, *annotated* by the values of V, i.e. there is one data structure for each $v \in V$. One can consider this as the *schema* of the associated set. An *instance* of A, w.r.t. a condition θ, is the collection of the data structures populated by S' elements, according to θ and v. The purpose is to compute aggregates over the data structures. An ASSET query consists of a base set of values V (i.e. a table) and successive definitions of associated sets, where aggregates of previously defined associated sets can be used for the definition of subsequent associated sets. An ASSET query is constructed incrementally, column-by-column, where a column corresponds to an associated set and is described by a data source, a defining condition and the aggregates to be computed over the associated set.

Evaluation of ASSET queries can be optimized in several ways. For example, an associated set and its aggregates can be computed either locally or remotely, where the data source (or partition of it) exists, by sending all the required data to the remote node; not materializing the associated set if the required aggregate computation is distributive or algebraic; build appropriate indexes on previously defined associated sets by analyzing the condition θ (e.g. build - or keep - it as an inverted list, if θ asks for membership into the associated set); choosing the most appropriate data structure to represent the associated set (B+ trees, min-max heaps, etc.); keep a single instance of identical associated sets of different rows of the base table, by creating a linked list within the new column. We have implemented most of these into our ASSET QE.

The first step in evaluating an ASSET query is to assign the associated sets into *computational rounds*: a computational round consists of independent to each other (directly or transitively) associated sets – there is a dependency between two associated sets if the one uses aggregates of the other (in its defining condition or aggregates). The output of a computational round is the base table for the next one. The data sources of the associated sets consist of data partitions. A computational round consists of one or more *basic computations*, one for each data partition of the computational round. It is a local computation of the associated sets using this data partition. To accommodate an optimization framework, we have developed two operators, one for the computational round and one for the basic computation:

- $AS_R(B, A_1(S_1),\ldots A_k(S_k))$, where B is the base table, $A_1,\ldots, A_k$ the associated sets that have to be computed in the computational round and $S_1,\ldots,S_k$ their respective data sources.

- $AS_B(B, S, A)$, where B is the base table, S a data partition and A the set of associated sets using this data partition.

This process is graphically depicted by Figure 2. The base table is sent to each data partition's node and the basic computation executes there. The whole process is coordinated by a main (coordinating) program. For example, consider Query Q2. Transactions table may be partitioned across several nodes (sources). The list of clientIDs (i.e. B_0) could be sent to all nodes, the partial F_is computed locally at the node and sent back to the coordinator, where they are concatenated to form F_is.

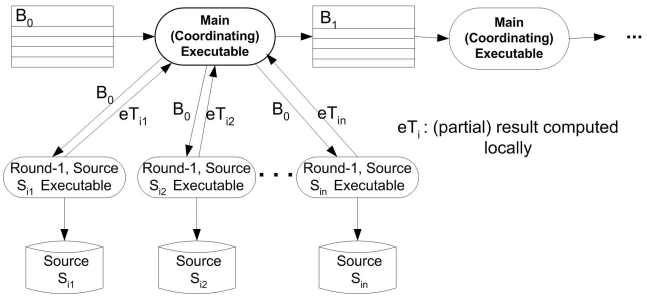

Figure 2: Evaluation of ASSET queries

This plain execution strategy can be generalized to a more elaborate one, where base tables are horizontally partitioned according to their estimated sizes and the computation of the ASSET query "flows" from left to right, possibly in parallel. The following algorithm describes this idea:

```
1: compute B0; round=1;
2: repeat {
3:    Partition B0 to B0^1, B0^2, …, B0^k;
4:    A={the set of assoc. sets of current round};
5:    B1 = AS_R^round(B0^1,A) U...U AS_R^round(B0^k,A);
6:    round++;
7:    B0 = B1;
8: } until (round>n);
```

Note that the combination step of line 5 can be omitted if partitioning remains the same from one computational round to the next. In general, one can think of an evaluation strategy represented by a graph with fork and join points at the end of computational rounds.

The complete architecture of our system is shown in Figure 3. An ASSET query is formulated either graphically using DataMingler or textually using an extended version of SQL . In both cases, an XML-based specification file is generated. Section 3 describes query formulation of ASSET queries. The XML-based specification file of an ASSET query is passed to the main parser (assetGenGlobal), which coordinates the execution of two lower-level parsers (assetGenRound and assetGenBasic) and produces the main (coordinating) C++ program. The assetGenRound parser assigns the associated sets to computational rounds and the assetGenBasic parser generates efficient C++ programs implementing the basic computations. The main C++ program manages the ASSET structure (the data structure representing the answer of the ASSET query) and coordinates the basic computations. All the generated C++ programs are then compiled and executed. Section 4 presents the ASSET Query Engine.

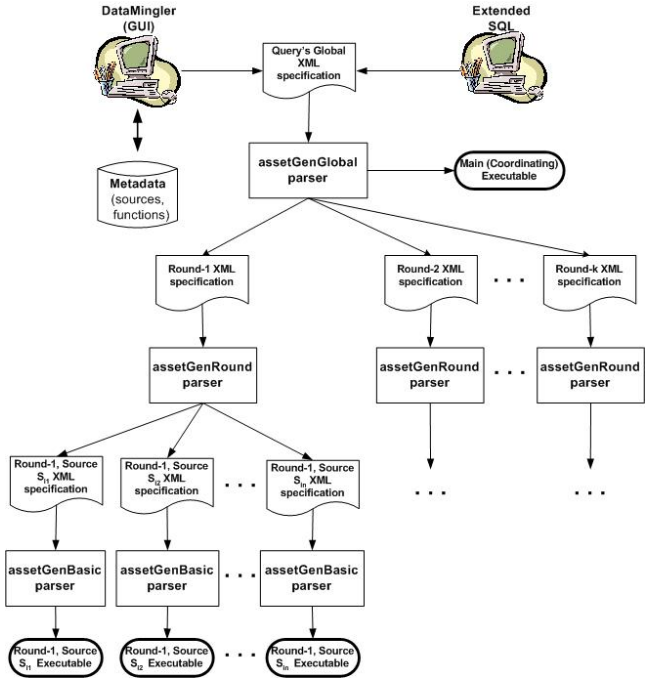

Figure 3: Expressing and evaluating ASSET queries

3. QUERY FORMULATION

ASSET queries can be formulated by either a GUI called DataMingler, or an extended SQL textual interface. Both generate an XML-based specification of the query that is fed to the ASSET QE.

3.1 Extended SQL

We follow the formalism of grouping variables [7]. 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> query. The proposed syntax is:

```
select A from R where θ group by A'
extended by A1(S1), A2(S2), … , An(Sn)
such that θ1, θ2, ..., θn
```

The selection list A may contain aggregate functions defined over the associated sets A_1, A_2, …, A_n. The answer

of the <select..from..where.. group by> SQL query serves as the base-values table. Condition θ_i involves attributes of the base-values table, constants and aggregates of associated sets $A_1,\ldots,A_{i-1}$., i=1,…n. For example, query Q2 can be expressed as (recall from Section 2 that *all(attr)* is an aggregate function returning the set of *attr* values of the involved table):

```
select clientID, Y.mostOften(categID)
from Clients
extended by X(Transactions), Y(Stocks)
such that X.clientID=clientID,
          Y.stockID in X.all(stockID)
```

3.2 DataMingler – A Spreadsheet-Like Tool

We have developed a spreadsheet-like GUI to manage data sources, user-defined aggregate functions and ASSET queries. It has been implemented in C++ code using the Qt4 C++ library for Windows from Trolltech. Since Qt is platform independent, DataMingler can be easily compiled for Unix/Linux operating systems.

3.2.1 Data Source Management

An ASSET query uses 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 we support databases (Postgres, MySQL, Oracle, SQL Server), flat files and socket-based streams. Figure 4 shows the first step in defining the Transactions data source.

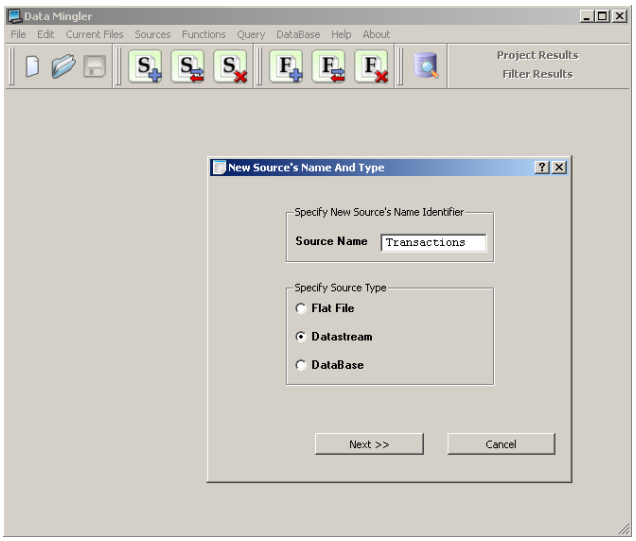

Figure 4: Defining a new data source in DataMingler

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 at several processing nodes.

3.2.2 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 (Figure 5).

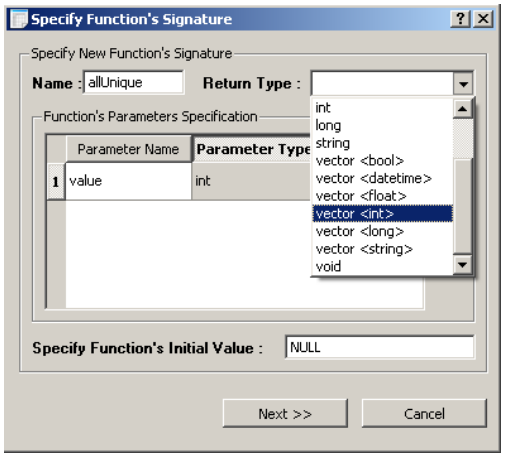

Figure 5: Defining a new aggregate function, allUnique, returning a set of distinct integers

The user 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++).

3.2.3 Asset Queries

Users specify ASSET Queries through DataMingler in a spreadsheet-like manner, column by column. The user initially specifies a base 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. Figure 6 shows the window for the definition of an associated set.

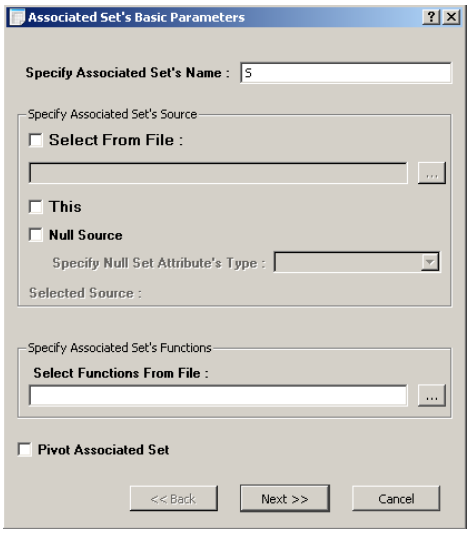

Figure 6: Building an ASSET query

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. DataMingler identifies dependencies, performs a topological sort and places them into "processing rounds", as required by the ASSET QE, explained in the following section.

4. The ASSET QUERY ENGINE (QE)

Once an ASSET query has been formulated and represented as an XML-based specification, it is passed to the ASSET QE for optimization, code generation and execution.

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* 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 (the "basic computational thread") to scan the data source and compute the associated sets related to that source. This 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 (the output of the ASSET query) 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 - currently has to be done manually, by altering the query's main C++ program.

5. PERFORMANCE

Figure 7 shows the performance of the query Q2 when the size of Transactions table varies from 100M to 600M records (15GB to 90GB) – all in one partition. We assumed 10M clients and 10K stocks. We tried to compare ASSET query engine's performance with standard SQL formulation using PostgreSQL but we could not get any results for even 200M records after 21 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.

6. CONCLUSIONS AND FUTURE WORK

In this paper we introduced ASSET queries, a useful class of data analysis queries that can be efficiently evaluated in distributed settings. While the basic component of an ASSET query, the associated set, is similar to a MapReduce operator, the ability to have multiple

associated sets, possibly correlated, placed next to each other in the same query, increases greatly the complexity of data analysis performed and offers significant optimization capabilities.

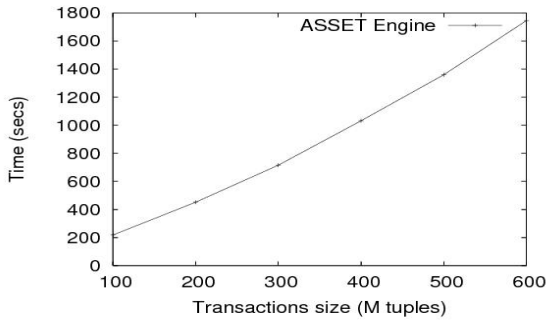

Figure 7: ASSET QE performance on query Q2

The main differences between ASSET queries and MapReduce can be summarized as: a) the data set of the mapping phase is defined using a condition, instead of using an arbitrary mapping function – less flexibility, more room for optimization, b) one may define multiple reduce sets for the same value, by using different defining conditions – this is not obvious how it can be done in *one* MapReduce, c) correlated aggregation ([6],[11]), an important topic in data analysis, can be expressed in one query instead of multiple, providing additional hints to the optimizer, and d) while overlapping reduce sets can be easily defined in ASSET queries by arbitrary conditions, in MapReduce special configuration of the mapping function is required.

We argued in [4] that ASSET queries' framework seems appropriate for expressing a significant class of data stream queries. In this case, data sources are continuous streams of data, associated sets are frequently represented as queues and the dependencies between associated sets dictate the order of update of the ASSET structure.

Current research involves theoretical work, improving implementation and benchmarking. We would like to: (a) develop a theoretical (algebraic) framework for associated sets, (b) extend our optimization platform, and (c) compare our performance results to those in [1] and [15]. In addition, it seems that a query language over the *powerset* of a relation (defining a set of subsets of the relation) could unify several proposals for ad hoc OLAP. We are investigating whether the associated sets framework can serve as the basis of such a language.

7. ACKNOWLEDGMENTS

This research work was partially supported by EU-ICT Grant ST-5-034957-STP. The authors would like to thank Prof. Diomidis Spinellis for pointing out useful references.

8. REFERENCES

[1] Abouzeid, A., et al. *HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Worksloads.* In VLDB, 2009, 922-933.

[2] Akinde, M. O., Bohlen, M. H., Johnson, T., Lakshmanan, L. V. S., Srivastava, D. *Efficient OLAP Query Processing in Distributed Data Warehouses*. In EDBT, 2002, 336-353.

[3] Bin, L. and Jagadish, HV: *A Spreadsheet Algebra for a Direct Data Manipulation Query Interface.* In ICDE, 2009, 417-428.

[4] Chatziantoniou, D., Pramatari, K., Sotiropoulos, Y. *COSTES: Continuous Spreadsheet-like Computations.* In Intern. Workshop on RFID Data Management (ICDE 2008).

[5] Chatziantoniou, D., Akinde, M., Johnson, T. and S. Kim, S. *The MD-Join: An Operator for Complex OLAP.* In Int. Conf. on Data Engineering (ICDE), 2001, 524-533.

[6] Chatziantoniou, D. and Ross, K. *Querying Multiple Features of Groups in Relational Databases*. In VLDB, 1996, 295-306.

[7] Chatziantoniou, D. *Using grouping variables to express complex decision support queries*. In DKE Journal, 61(1), 2007, 114-136.

[8] Cieslewicz, J., Berry, J., et al. *Realizing parallelism in database operations*. In DaMoN, 2006.

[9] Dean, J. and Ghemawat S. *MapReduce: Simplified Data Processing on Large Clusters*. In 6th Symp. on Operating System Design and Implementation, 2004, 137-150.

[10] 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

[11] Gehrke, J., Korn, F. and Srivastava, D. *On Computing Correlated Aggregates Over Continual Data Streams.* In SIGMOD Conference, 2001, 13-24.

[12] Hive Project. `http://hadoop.apache.org/hive/`

[13] Mamoulis, N. *Efficient Processing of Joins on Set-valued Attributes.* In ACM SIGMOD, 2003, 157-168.

[14] Olston, C., Reed, B., Srivastava, U., Kumar, R. and Tomkins, A. *Pig latin: a not-so-foreign language for data processing.* In ACM SIGMOD, 2008, 1099-1110.

[15] Pavlo, A., et al. *A Comparison of Approaches to Large-Scale Data Analysis.* In ACM SIGMOD, 2009, 165-178.

[16] Steenhagen, HJ., Apers, P., Blanken, HM. *Optimization of nested queries in a complex object model*. In EDBT, 1994, 337-350.

[17] Stonebraker, M., et al. *C-Store: A Column-oriented DBMS.* In Intern. Conf. on VLDB, 2005, 553-564.

[18] Witkowski, A., Bellamkonda, S., et al. *Spreadsheets in RDBMS for OLAP.* In SIGMOD, 2003, 52-63.