

ATLaS: A Native Extension of SQL for Data Mining

Haixun Wang
IBM T. J. Watson Research Center
Hawthorne, NY 10532
haixun@us.ibm.com

Carlo Zaniolo
Computer Science Dept, UCLA
Los Angeles, CA 90095
zaniolo@cs.ucla.edu

Abstract

A lack of power and extensibility in their query languages has seriously limited the generality of DBMSs and hampered their ability to support data mining applications. Thus, there is a pressing need for more general mechanisms for extending DBMSs to support efficiently database-centric data mining applications. To satisfy this need, we propose a new extensibility mechanism for SQL-compliant DBMSs, and demonstrate its power in supporting decision support applications. The key extension is the ability of defining new table functions and aggregate functions in SQL—rather than in external procedural languages as Object-Relational (O-R) DBMSs currently do. This simple extension turns SQL into a powerful language for decision-support applications, including ROLAPs, time-series queries, stream-oriented processing, and data mining functions. First, we discuss the use of ATLaS for data mining applications, and then the architecture and techniques used in its realization.

1 Introduction

The inadequate power and extensibility of SQL has limited DBMSs' generality and ability to support an ever growing list of new applications. Substantial extensions have been added to SQL over the years, including those for recursive queries, active rules, ROLAPs, and datablades; yet data mining applications remain an unsolved challenge for DBMSs [2]. In this paper, we propose minimal extensions of SQL that are particularly effective at expressing efficiently data mining tasks; these extensions are user-defined aggregates and tables functions written in SQL itself and, for improved performance, main-memory tables.

ATLaS adds to SQL the ability of defining new User Defined Aggregates and Table Functions. (ATLaS is an acronym for 'Aggregate & Table Language and System'.) The User Defined Aggregates (UDAs) of ATLaS implement a stream-oriented computation model, whereby UDAs accept a stream as input and produce a stream as output. Our UDAs can express both traditional blocking aggregates and non-blocking aggregates—such as online aggregates [8] and the continuous aggregates used for time series [17]—in a syntactic framework that makes it easy to identify nonblocking aggregates. Finally, UDAs and table functions are de-

finied in SQL itself—rather than in the external procedural languages required for defining new functions in current O-R systems. This closure property is the source of great power and flexibility, and it is also conducive to a stream-oriented computation model. The ability of expressing data mining functions in SQL involves a modest performance overhead over the same applications written in C/C++ (and an improvement over Java or PL/SQL). We prove this by an ATLaS implementation of the Apriori algorithm, i.e., by applying the litmus test proposed in [2] for database-centric data mining.

Besides being particularly effective for data mining tasks, these extensions are also beneficial in other application domains, inasmuch as they turn SQL into a Turing-complete language [14]. In particular, spatial and temporal applications were discussed in [4] and data stream queries were studied in [23].

In the next section, we discuss related work, and in Section 3 we introduce ATLaS by examples. In Section 4 we discuss recursion and table functions, and in Section 5 we discuss ROLAPs, while data mining is discussed in Section 6. Performance and architecture are presented in Sections 7 and 8.

2 Previous Work

A significant amount of research has focused on the problem of overcoming the limitations of database systems in data mining tasks. One first line of research attempts to support mining tasks through suitable extensions of the database query language [6, 15, 9]. The wide range of tasks to be supported and the lack of efficient implementation techniques proved serious challenges for these approaches, leading many to believe that general extensions of SQL for data mining might be difficult to achieve [18]. Therefore, in [18] the more basic question was investigated of whether it was possible for a team of experts to produce an efficient implementation of Apriori on a state-of-the art O-R DBMSs. A reasonable level of performance was achieved through the use of specialized join techniques and user-defined functions (UDFs); these, however, were the source of many difficulties in programming and debugging [18]. While it has been

shown that many of these problems can be solved by starting from a different data representation [16], and more efficient algorithms have been later developed [7], the Apriori algorithm provides a well-studied case study of the difficulty of implementing a rather simple data mining algorithm to SQL-compliant DBMSs, and will thus be used in this paper.

The problems with SQL and commercial systems encountered by these approaches motivated interest in using other database languages. For instance, an algebra based on the 3W model is proposed in [13] to describe and unify the intensional, extensional, and data dimensions of the mining process [13]. In [3, 5], a logic-based approach is proposed that integrates the inductive and deductive aspects of the mining process using the $\mathcal{LDL}++$ language and exploiting the UDA capabilities of that language [21]. We have incorporated into ATLaS many of the lessons we learned from $\mathcal{LDL}++$.

We now introduce ATLaS by examples; a more complete discussion can be found in [1].

3 User-Defined Aggregates

ATLaS adopts SQL-3 idea of specifying UDAs by an *initialize*, an *iterate*, and a *terminate* computation [11, 12], but then uses the AXL approach [22] of expressing these three computations by a single procedure written in SQL—rather than three written in procedural languages as in SQL-3¹.

Example 1 defines an aggregate equivalent to the standard **avg** aggregate in SQL. The first line of this aggregate function declares a local table, **state**, to keep the sum and count of the values processed so far. While, for this particular example, **state** contains only one tuple, it is in fact a table that can be queried and updated using SQL statements and can contain any number of tuples (see later examples). These SQL statements are grouped into the three blocks labelled respectively INITIALIZE, ITERATE, and TERMINATE. Thus, INITIALIZE inserts the value taken from the input stream and sets the count to 1.

EXAMPLE 1. *Defining the standard aggregate average*

```
AGGREGATE myavg(Next Int) : Real
{ TABLE state(sum Int, cnt Int);
  INITIALIZE : {
    INSERT INTO state VALUES (Next, 1);
  }
  ITERATE : {
    UPDATE state SET sum=sum+Next, cnt=cnt+1;
  }
  TERMINATE : {
    INSERT INTO RETURN SELECT sum/cnt FROM state;
  }
}
```

¹Although UDAs have been left out of SQL 1999 specifications, they were part of early SQL-3 proposals, and supported by some commercial DBMS [10].

The ITERATE statement updates the table by adding the new input value to the sum and 1 to the count. The TERMINATE statement returns the final result(s) of computation by INSERT INTO RETURN (to conform to SQL syntax, RETURN is treated as a virtual table; however, it is not a stored table and cannot be used in any other role).

Observe that the three SQL statements in the INITIALIZE, ITERATE, and TERMINATE groups play the same role as the external functions with the same names in SQL-3 aggregates. But here we have assembled the three functions under one procedure, thus supporting the declaration of their shared tables (the **state** table in this example). This table is allocated just before the INITIALIZE statement is executed and deallocated just after the TERMINATE statement is completed.

This approach to aggregate definition is very general. For instance, say that we want to support online aggregation [8], an important concept not considered in SQL-3. Since averages converge to a final value well before all the tuples in the set have been visited, we can have an online aggregate that returns the average-so-far every, say, 200 input tuples. In this way, the user or the calling application can stop the computation as soon as convergence is detected.

Thus in Example 2, we have removed the statements from TERMINATE and added a RETURN statement to ITERATE². The UDA **online_avg**, so obtained, takes a stream of values as input and returns a stream of values as output (one every 200 tuples). While each execution of the RETURN statement produces here only one tuple, in general, it can produce (a stream of) several tuples. Thus UDAs operate as general stream transformers.

EXAMPLE 2. *Online averages*

```
AGGREGATE online_avg(Next Int) : Real
{ TABLE state(sum Int, cnt Int);
  INITIALIZE : {
    INSERT INTO state VALUES (Next, 1);
  }
  ITERATE : {
    UPDATE state SET sum=sum+Next, cnt=cnt+1;
    INSERT INTO RETURN
      SELECT sum/cnt FROM state
      WHERE cnt % 200 = 0;
  }
  TERMINATE : { }
```

ATLaS uses the same basic framework to define both blocking and non-blocking aggregates. A typical SQL aggregate is blocking: it is unable to produce the first tuple of the output until it has seen the entire input. The presence of a non-blocking aggregate is clearly denoted by the fact that the TERMINATE clause is either empty or absent.

²ATLaS syntax allows user to omit writing a TERMINATE altogether; but then the system inserts a 'TERMINATE: { }' clause that, e.g., causes the deallocation of all local tables

The most common operational semantics for SQL aggregates in commercial systems is that the data is first sorted according to the GROUP-BY attributes: thus the very first operation in the computation is a blocking operation. However, ATLaS's default semantics for UDAs is that the data is pipelined through the INITIALIZE and ITERATE clauses where the input stream is transformed into the output stream: the only blocking operations (if any) are those specified in TERMINATE, and only take place at the end of the computation. Therefore, even for blocking aggregates ATLaS performs most of the computation in a stream oriented fashion.

ATLaS default semantics leads to a (nonblocking) hash-based implementation for UDAs. However, (blocking) sort-based aggregates are also supported; in fact they can be easily specified by adding an ORDER BY clause to the SELECT target list, as described in Algorithm 3.

UDAs are called as any other builtin aggregate. For instance, given a database table employee(Eno, Name, Sex, Dept, Sal), the following statement computes the average salary of employees in department 1024 by their gender:

```
SELECT Sex, online_avg(Sal)
FROM employee WHERE Dept=1024 GROUP BY Sex;
```

Thus the results of the selection, defined by Dept=1024, are pipelined to the aggregate in a stream-like fashion. In general, we can view ATLaS programs as consisting of two kinds of stream-oriented computations: one is the computation of select-project-join expressions, whose order of execution is largely controlled by the system, and whose results are passed to the UDAs, where the computation is more closely controlled by their user-written definitions.

While this topic is beyond the scope of this paper, it is important to observe that ATLaS UDAs have a formal semantics based on Datalog's fixpoint semantics [21]. This prescribes that when multiple aggregate functions are used in the same select list, the Cartesian product of their individual results must be returned.

In Example 3, we define a **minpair** aggregate that returns the point where a minimum occurs along with its value at the minimum.

EXAMPLE 3. *Define minpair in ATLaS*

```
AGGREGATE minpair(iPoint Int, iValue Int): (mPoint Int, mValue Int)
{
  TABLE mvalue(value Int);
  TABLE mpoints(point Int);
  INITIALIZE: {
    INSERT INTO mvalue VALUES (iValue);
    INSERT INTO mpoints VALUES(iPoint);
  }
  ITERATE: {
    UPDATE mvalue SET value = iValue
      WHERE iValue < value;
    DELETE FROM mpoints WHERE SQLCODE = 0;
    INSERT INTO mpoints
      SELECT iPoint FROM mvalue
      WHERE iValue =mvalue.value;
```

```

  }
  TERMINATE: {
    INSERT INTO RETURN
      SELECT point, value FROM mpoints, mvalue;
  }
}
```

Here we use two tables: **mvalue** table holds, as its only entry, the current min value, while **mpoints** holds all the points where this value occurs. In the ITERATE statement we have used the SQLCODE to 'remember' if the previous statement updated **mvalue**, since the old value was larger than the new one and the old points must be discarded. Indeed, SQLCODE is a convenient labor-saving device of standard SQL that is set to 0 if the last SELECT, UPDATE, or DELETE statement was *successful*—i.e., if more than zero tuples were, respectively, selected, updated, or deleted by its execution. Then, the last statement in ITERATE adds the new **iPoint** to **mpoints** if the input value is equal to the current min value. In the UDA definition the formal parameters of the UDA function are treated as constants in the SQL statements. Thus, this third INSERT statement adds the constant value of **iPoint** to the **mpoints** relation, provided that **iValue** is the same as the value in **mvalue**—thus the FROM and WHERE clauses operate here as conditionals. The RETURN statement returns the final list of min pairs as a stream.

The set of results returned by the UDA of Example 3 is independent from the particular order in which data is streamed through the UDA. But the stream-oriented computation model of UDAs makes it easy to take advantage of the order in which the data is being processed. Frequently, this order is of semantic significance. For instance, temporal database information is often organized chronologically. A well-known problem in temporal databases is how to support the operation of interval coalescing. Example 4 shows how to implement this computation in ATLaS.

EXAMPLE 4. *Coalescing*

```
AGGREGATE coalesce(from TIME, to TIME): (start TIME, end TIME)
{
  TABLE state(cFrom TIME, cTo TIME);
  INITIALIZE: {
    INSERT INTO state VALUES(from,to);
  }
  ITERATE: {
    UPDATE state SET cTo = to
      WHERE cTo >= from AND cTo < to;
    INSERT INTO RETURN SELECT cFrom, cTo
      FROM state WHERE cTo < from;
    UPDATE state SET cFrom = from, cTo = to
      WHERE cTo < from;
  }
  TERMINATE: {
    INSERT INTO RETURN
      SELECT cFrom, cTo FROM state;
  }
}
```

Our UDA **coalesce** takes two parameters: **from** is the start time, **to** is the end time. Under the assumption that tuples

ATLaS-program	:	ATLaS-dcl* SQL-statements*
ATLaS-dcl	:	Table-dcl UDA-dcl TableFunc-dcl
UDA-dcl	:	AGGREGATE ID (Type) : (Type) { ATLaS-dcl* INITIALIZE : [{ SQL-statement* }] ITERATE : { SQL-statement* } [TERMINATE : { SQL-statement* }] }
TableFunc-dcl	:	FUNCTION ID (Type) : (Type) { ATLaS-dcl* SQL-statement* }

Table 1: Syntax of ATLaS extensions

are sorted by increasing start time, then we can perform the task in one scan of the data. In the ITERATE routine, when the new interval overlaps the current interval kept in the **state** table, we merge the two intervals. Otherwise, the current interval is returned while the new interval becomes the current one.

ATLaS Syntax Table 1 summarizes the extension of ATLaS added to SQL. Basically, an ATLaS program starts with a set of declarations (for tables, UDAs and table functions) and proceeds with a sequence of SQL statements (that use those tables, UDAs and table functions). The declaration of UDAs and table functions, then consists of (i) the declaration of their types and formal parameters, (ii) the declaration of local tables, UDAs and table functions, and (iii) a sequence of SQL statements; for UDAs, those SQL statements are clustered in the three groups INITIALIZE, ITERATE and TERMINATE. Table functions are discussed in more details in the next section. The complete definition for the ATLaS syntax can be found in [1].

4 Table Functions and Recursive UDAs

Example 5 illustrates the typical structure of an ATLaS program. The declaration of the table **dgraph(start, end)** is followed by that of the UDA **reachable**; the table and the UDA are then used in an SQL statement that calls for the computation of all nodes reachable from node '000'. The clause **SOURCE(mydb)** denotes that **dgraph** is a table from a database that is known to the systems by the name 'mydb'. (Without the **SOURCE** clause **dgraph** is local to the program and discarded once its execution completes).

The program of Example 5 shows one way in which the transitive closure of a graph can be expressed in ATLaS. We use the recursive UDA **reachable** that perform a depth-first traversal of the graph by recursively calling itself. Upon receiving a node, **reachable** returns the node to the calling

query along with all the nodes reachable from it. (We assume that the graph contains no directed cycle; otherwise a table will be needed to memorize previous results and avoid infinite loops [1].)

EXAMPLE 5. *Computation of Transitive Closure*

```
TABLE dgraph(start Char(10), end Char(10)) SOURCE ('mydb');
AGGREGATE reachable(Rnode Char(10)) : Char(10)
{
  INITIALIZE: ITERATE: {
    INSERT INTO RETURN VALUES (Rnode);
    INSERT INTO RETURN
      SELECT reachable(end) FROM dgraph
      WHERE start=Rnode;
  }
}
SELECT reachable(dgraph.end) FROM dgraph
WHERE dgraph.start='000';
```

Observe that the INITIALIZE and ITERATE routine in Example 5 share the same block of code.

Besides the Prolog-like top-down computation of Example 5, we can also express easily the bottom-up computation used in Datalog [1], and a stream-oriented computation will be discussed in the next section.

Recursive queries can be supported in ATLaS without any new construct since UDAs can call other aggregates or call themselves recursively. Examples of application of recursive UDAs in data mining will be discussed later.

Along with recursive aggregates, table functions defined in SQL play a critical role in expressing data mining applications in ATLaS. For instance, let us consider the table function **dissemble** that will be used to express decision tree classifiers. Take for instance the well-known Play-Tennis example of Table 2; here we want to classify the value of **Play** as a 'Yes' or a 'No' given a training set such as that shown in Table 2.

The first step for most scalable classifiers [19] is to convert the training set into column/value pairs. This conver-

RID	Outlook	Temp	Humidity	Wind	Play
1	Sunny	Hot	High	Weak	No
2	Sunny	Hot	High	Strong	No
3	Overcast	Hot	High	Weak	Yes
4	Rain	Mild	High	Weak	Yes
5	Rain	Cool	Normal	Weak	Yes
6	Rain	Cool	Normal	Strong	Yes
7	Overcast	Cool	Normal	Strong	No
8	Sunny	Mild	High	Weak	No
9	Sunny	Cool	Normal	Weak	Yes
10	Rain	Mild	Normal	Weak	Yes
11	Sunny	Mild	Normal	Strong	Yes
12	Overcast	Mild	High	Strong	Yes
13	Overcast	Hot	Normal	Weak	Yes
14	Rain	Mild	High	Strong	No

Table 2: The relation **PlayTennis**

sion, although conceptually simple, is hard to express succinctly in SQL. Consider the **PlayTennis** relation as shown in Table 2. We want to convert **PlayTennis** into a new stream of three columns (**Col**, **Value**, **YorN**) by breaking down each tuple into four records, each record representing one column in the original tuple, including the column number, column value and the class label **YorN**. We can define the table function **dissemble** of Example 6 to solve the problem. Then, using this table function and the recursive aggregate **classify**, Algorithm 1 implements a scalable decision tree classifier using merely 15 statements.

EXAMPLE 6. *Dissemble a relation into column/value pairs.*

```

FUNCTION dissemble(v1 Int, v2 Int, v3 Int, v4 Int, YorN Int)
: (col Int, val Int, YorN Int);
{
  INSERT INTO RETURN VALUES (1, v1, YorN), (2, v2, YorN),
    (3, v3, YorN), (4, v4, YorN);
}

```

In the INITIALIZE and ITERATE statements of **classify** in Algorithm 1, we update the class histogram kept in the **summary** table for each column/value pair. The TERMINATE routine first computes the gini index for each column using the histogram. However, if a column has a single value ($\text{count}(\text{Value}) \leq 1$), or tuples in the partition belongs to one class ($\text{sum}(\text{Yc})=0$ or $\text{sum}(\text{Nc})=0$), then the column is not splittable and hence, not inserted into **ginitable**. On line 12, we select the splitting column which has the minimal gini index. A new sub-branch is generated for each value in the column. The UDA **minpair**, defined in Example 3, returns the minimal gini index as well as the column where the minimum value occurred. After recording the current split into the **result** table, we call the classifier recursively to further classify the sub nodes. On line 14, GROUP BY **m.Value** partitions the records in **treenodes** into **MAXVALUE** subnodes, where **MAXVALUE** is the largest number of different values in any of the table columns (three for Table 2). The recursion terminates if table **mincol** is empty, that is, there is no valid

Algorithm 1 A Scalable Decision Tree Classifier

```

1: AGGREGATE classify(iNode Int, RecId Int, iCol Int,
  iValue Int, iYorN Int)
2: { TABLE treenodes(RecId Int, Node Int, Col Int,
  Value Int, YorN Int);
3: TABLE mincol(Col Int);
4: TABLE summary(Col Int, Value Int, Yc Int, Nc Int)
  INDEX (Col, Value);
5: TABLE ginitable(Col Int, Gini Int);
6: INITIALIZE : ITERATE : {
7:   INSERT INTO treenodes
    VALUES(RecId, iNode, iCol, iValue, iYorN);
8:   UPDATE summary
    SET Yc=Yc+iYorN, Nc=Nc+1-iYorN
    WHERE Col = iCol AND Value = iValue;
9:   INSERT INTO summary
    SELECT iCol, iValue, iYorN, 1-iYorN
    WHERE SQLCODE<<0;
  }
10: TERMINATE : {
11:   INSERT INTO ginitable
    SELECT Col, sum((Yc*Nc)/(Yc+Nc))/sum(Yc+Nc)
    FROM summary GROUP BY Col;
    HAVING count(Value) > 1
    AND sum(Yc)>0 AND sum(Nc)>0;
12:   INSERT INTO mincol
    SELECT minpair(Col, Gini)→mPoint FROM ginitable;
13:   INSERT INTO result
    SELECT iNode, Col FROM mincol;
    /* Call classify() recursively to partition each of its */
    /* subnodes unless it is pure. */
14:   SELECT classify(t.Node*MAXVALUE+m.Value+1,
    t.RecId, t.Col, t.Value, t.YorN)
    FROM treenodes AS t,
    ( SELECT tt.RecId RecId, tt.Value Value
    FROM treenodes AS tt, mincol AS m
    WHERE tt.Col=m.Col) AS m
    WHERE t.RecId = m.RecId
    GROUP BY m.Value;
  }
15: }

```

column to further split the partition.

To classify the **PlayTennis** dataset shown in Table 2, we use the following statement

```

SELECT classify(0, p.RID, d.Col, d.Val, d.YorN)
FROM PlayTennis AS p,
  TABLE(dissemble(Outlook,Temp, Humidity, Wind, Play)) AS d;

```

Table functions and recursion are also supported in SQL 1999, but, at the best of our knowledge, there is no simple way to express decision-tree classifiers in SQL (or for that matter in Datalog). Thus the fact that a concise expression for this algorithm is now possible suggests that ATLaS benefits from additional sources of expressive power. In the next section we find that this is connected with the stream-oriented computation model used by ATLaS, which, in turn, relates to the nonblocking behavior of certain aggregates, such as online averages.

5 ROLAPs

Powerful aggregate extensions based on modifications and generalization of group-by constructs have recently been proposed by researchers, OLAP vendors, and standard committees. New operators, such as ROLLUP and CUBE, have been included in SQL-3 and implemented in major commercial DBMSs. We will now express these extensions in ATLaS.

The purpose of ROLLUP is to create subtotals at multiple detail levels from the most detailed one, up to the grand total. This functionality could be expressed in basic SQL by combining several SELECT statements with UNIONS. For instance, to roll up a sales table along dimensions such as Time, Region, and Department, we can use the query of Example 7.

EXAMPLE 7. *Using Basic SQL to express ROLLUP*

```
SELECT Time, Region, Dept, SUM(Sales)
FROM Sales GROUP BY Time, Region, Dept
UNION ALL
SELECT Time, Region, 'all', SUM(Sales)
FROM Sales GROUP BY Time, Region
UNION ALL
SELECT Time, 'all', 'all', SUM(Sales)
FROM Sales GROUP BY Time
UNION ALL
SELECT 'all', 'all', 'all', SUM(Sales)
FROM Sales;
```

The problem with the approach in Example 7, above, is that each of the four SELECT statements could result in a new scan of the table, even though all needed subtotals can be gathered in a single pass. Thus, a new ROLLUP construct was introduced in SQL.

No ad hoc operator is needed in ATLaS to express rollup queries. For instance, in ATLaS the above query can be expressed succinctly as follows:

```
SELECT rollup(Time, Region, Dept, Sales) FROM data;
```

Indeed, the rollup functionality can be expressed by ATLaS in several different ways. Algorithm 2 shows an implementation of the **rollup** aggregate used in the above query, where the dataset is assumed ordered by Time, Region, and Dept.

Algorithm 2 combines UDAs and table functions to implement **rollup**. We use the following in-memory table (as indicated by keyword MEMORY in the code) to keep track of the subtotals at each rollup level j ($j = 1, \dots, 4$, with level 4 corresponding to the grand total):

```
memo(j Int, Time Char(20), Region Char(20),
      Dept Char(20), Sum Real)
```

At the core of the algorithm, we have the four entries added to **memo** by the INITIALIZE statement (line 8). The first entry has rollup level one and its subtotal (last column) is initialized to the sales amount of the first record. The

Algorithm 2 Roll-up sales on Time, Region, Dept

```
1: AGGREGATE rollup(T Char(20), R Char(20), D Char(20), Sales
   Real) : (T Char(20), R Char(20), D Char(20), Sales Real)
2: { TABLE memo(j Int, Time Char(20), Region Char(20),
   Dept Char(20), Sum Real) MEMORY;
3:   FUNCTION onestep(L Int, T Char(20), R Char(20),
   D Char(20), S Real)
   : (T Char(20), R Char(20), D Char(20), Sales Real)
4:   { INSERT INTO RETURN
   SELECT Time, Region, Dept, Sum FROM memo
   WHERE L=j
   AND (T<>Time OR R<>Region OR D<>Dept);
5:   UPDATE memo SET Sum = Sum + (SELECT m.Sum
   FROM memo AS m WHERE m.j=L)
   WHERE SQLCODE=0 AND j=L+1;
6:   UPDATE memo
   SET Time=T, Region=R, Dept=D, Sum=S
   WHERE SQLCODE=0 AND j=L;
   }
7:   INITIALIZE: {
8:   INSERT INTO memo
   VALUES (1, T, R, D, Sales), (2, T, R, 'all', 0),
   (3, T, 'all', 'all', 0), (4, 'all', 'all', 'all', 0);
   }
9:   ITERATE: {
10:  UPDATE memo SET Sum = Sum + Sales
   WHERE Time=T AND Region=R AND Dept=D;
11:  INSERT INTO RETURN SELECT os.*
   FROM TABLE(onestep(1, T, R, D, Sales)) AS os
   WHERE SQLCODE <> 0;
12:  INSERT INTO RETURN SELECT os.*
   FROM TABLE(onestep(2, T, R, 'all', 0)) AS os
   WHERE SQLCODE = 0;
13:  INSERT INTO RETURN SELECT os.*
   FROM TABLE(onestep(3, T, 'all', 'all', 0)) AS os
   WHERE SQLCODE = 0;
   }
14:  TERMINATE: {
15:  INSERT INTO RETURN SELECT os.*
   FROM TABLE(onestep(1, 'all', 'all', 'all', 0)) AS os;
16:  INSERT INTO RETURN SELECT os.*
   FROM TABLE(onestep(2, 'all', 'all', 'all', 0)) AS os;
17:  INSERT INTO RETURN SELECT os.*
   FROM TABLE(onestep(3, 'all', 'all', 'all', 0)) AS os;
18:  INSERT INTO RETURN
   SELECT Time, Region, Dept, Sum
   FROM memo WHERE j=4;
   }
19: }
```

subtotals for the other three entries are set to zero. Let $memo_j$ denote the memo tuple at level j ; then, $memo_j$ contains $j - 1$ occurrences of 'all'.

The four SQL statements in the ITERATE group (i) determine the rollup levels to which the next tuple in the input will contribute, and (ii) for each such level, return values, and update the memo table. For instance, if the three leftmost columns of the new input tuple match those of $memo_1$, then the new input value is also of level one. If this is not the case, but the two leftmost columns match

those of $memo_2$, then the new tuple is of level two, and so on. If one (none) of the columns matches, then the new tuple is considered to be of level 3 (level 4). For incoming tuples at level 1, we update the subtotal for $memo_1$ but return no result. For tuples of level 2 (level 3), we return the current subtotal from $memo_1$ (and $memo_2$), reset this subtotal using the input value, and then update the subtotal at $memo_2(memo_3)$. For tuples belonging to level 4, we return the subtotals from $memo_j, j = 1, 2, 3$ and reset them to new input value. The TERMINATE statement returns the subtotals from $memo_j, j = 1, 2, 3, 4$ where $memo_4$ now contains the sum of all sales.

This computation is implemented by Algorithm 2 with the help of a special variable of SQL, SQLCODE. If no updates are made on line 10, i.e., if SQLCODE \neq 0, we need to use `onestep()` to “roll up” the subtotals from level 1 to level 2 (line 11). If the roll-up is successful, then we need to check if further roll-ups from level 2 to level 3, and then from level 3 to level 4 are necessary.

The table function `onestep` is rather simple. We first test if the level of the record being passed is different from the entry $memo_j$ (to simplify this test some of its columns are conveniently set to `all`). If they are different, then the subtotal for stored in $memo_j$ must be returned. This subtotal must also be passed (‘rolled-up’) to the next level: i.e., to level $j + 1$. Finally, the subtotal at $memo_j$ must be reset from current input record to restart aggregation on a new set of group-by columns.

In Algorithm 2, we assumed that the data is already sorted on the rollup columns. When this is not the case, then we use the UDA `sort_and_roll`, of Algorithm 3, which first sort the data and then calls the UDA rollup. Furthermore, the CUBE operator is easily implemented by a sequence of three sort-and-roll [1].

In Algorithm 2, `sort_temp` applies the standard SQL clause ORDER BY to the output tuples. Clearly, an SQL statement that contains an ORDER BY clause is blocking, since it requires sorting. Therefore, the table function `sort_temp` is also blocking since it contains this statement. Thus, table functions can become blocking because they contain SQL-statements with ORDER BY, or blocking aggregates or blocking table functions; but, except for those situations, table functions are nonblocking.

6 The Apriori Algorithm

Previous work on database-centric data mining applications has shown that these are not supported well by current O-R systems, and there is no clear understanding on which SQL extensions are needed to solve the problem. In elucidating this sorry state of affairs the award winning paper [18] also established the Apriori algorithm as the litmus test that any aspiring solution must satisfy. The AXL system [22] failed this acid test—also all the applications presented

Algorithm 3 Sort and roll sales on Time, Region, Dept

```

1: AGGREGATE sort_and_roll(T Char(20), R Char(20), D Char(20),
   Sales Real) : (T Char(20), R Char(20), D Char(20), Sales Real)
2: { TABLE temp(A1 Char(20), A2 Char(20), A3 Char(20), V Real)
      MEMORY;
3: FUNCTION sort_temp()
      : (A1 Char(20), A2 Char(20), A3 Char(20), V Real)
4: { INSERT INTO RETURN
      SELECT * FROM temp ORDER BY A1, A2, A3;
  }
5: INITIALIZE: ITERATE: {
      INSERT INTO temp VALUES (T, R, D, Sales);
  }
6: TERMINATE: {
7: INSERT INTO RETURN
      SELECT rollup(t.A1, t.A2, t.A3, t.V)
      FROM TABLE (sort_temp()) AS t;
  }
8: }
```

in Section 4 and some of those discussed in Section 3 could not be supported in AXL. These setbacks helped us identifying important features that were missing from AXL and various aspects of its implementation architecture and query optimizer that required major improvements. The new features added to ATLaS include support for (i) table functions coded in SQL, (ii) in-memory tables, (iii) OIDs used to reference tuples and implement (in-memory) data structures, and (iv) many changes in the optimizer to improve the execution speed of programs. These improvements have produced the ATLaS system that supports efficiently a wide spectrum of data-intensive applications, including the Apriori algorithm.

Problem Statement. The problem of mining frequent itemsets over basket data was introduced by R. Agrawal et al. in [2]. Let $I = \{i_1, \dots, i_m\}$ be a set of literals, called items. Let D be a set of transactions, where each transaction T is a set of items. We say that a transaction T contains itemset X , if $T \supseteq X$. Itemset X has support s in the transaction set D if no less than s transactions in D contain X . Given a set of transactions D , the problem of mining frequent itemsets is to generate all itemsets that have support greater than the user-specified minimum support (called *MinSup*).

Data Organization. Let a transaction dataset be represented by a stream of items, and each item is encoded with an integer $t, t > 0$. Adjacent transactions in the stream are separated by a special symbol, 0. Within each transaction, items are sorted by their integer value. For example, the following stream represents a dataset of 5 transactions:

$$(6.1) \quad 0, 2, 3, 4, 0, 1, 2, 3, 4, 0, 3, 4, 5, 0, 1, 2, 5, 0, 2, 4, 5$$

Thus, we assume that this stream is drawn from a database table: `baskets(item INT)`.

However, our algorithm does not depend on the existence of such a table, since it will work for data taken from a

Algorithm 4 Main ATLaS Program for Apriori

```

1: TABLE baskets(item Int) SOURCE 'marketdata';
2: TABLE trie(item Int, father REF(trie))
   INDEX(father) MEMORY;
3: TABLE candS(item Int, trieref REF(trie), freqcount Int)
   INDEX(cit, trieref) MEMORY;
4: TABLE fitems(item Int) INDEX(item);
   /* generat frequent one-itemsets */
5: INSERT INTO fitems
   SELECT item FROM baskets WHERE item > 0
   GROUP BY item HAVING count(*) ≥ MinSup;
   /* initialize the trie to contain frequent one-itemsets */
6: INSERT INTO trie SELECT item, null FROM fitems;
   /* self-join frequent 1-itemsets to candidate 2-itemsets */
7: INSERT INTO candS
   SELECT t1.itno, t2.OID, 0 FROM trie AS t1, trie AS t2
   WHERE t1.itno > t2.itno;
   /* Generate (k+1)-itemsets from k-itemsets (k=2,...) */
8: SELECT countset(item, 2, MinSup, candS) FROM baskets;

```

stream, a view, or generated by a query.

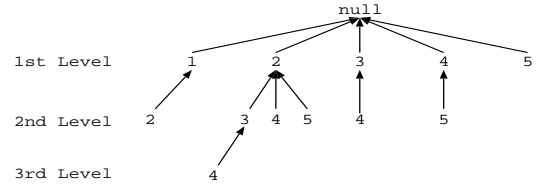
We use a prefix tree, or a trie, to store frequent itemsets. An example trie is shown in Figure 1(a). Each node in the trie represents a frequent itemset that contains all the items on the path from the root to that node. For instance, the only frequent 3-itemset in Figure 1(a) is (2,3,4). In ATLaS, the trie is represented by the in-memory table **trie**, where each record contains an item, as well as a pointer to its parent node, which is another record in the **trie** table: **trie(itno INT, father REF(trie))**.

The trie grows level by level. To find frequent $(k + 1)$ -itemsets, we first generate candidates based on the k -itemsets. The candidates are stored in an in-memory table: **candS(cit Int, trieref REF(trie), freqcount Int)**.

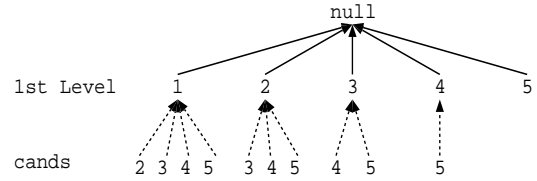
Each record in **candS** contains an item, **cit**, and a reference, **trieref**, which points to a leaf node of the trie. If the leaf node is on level k , then **cit**, together with the frequent itemset referenced by **trieref**, represents a candidate itemset of $k + 1$ items. The support of the candidate, **freqcount**, is updated in the algorithm as we count its occurrence. For efficiency purposes, both **trie** and **candS** are indexed. More specifically, **trie** is indexed on **father**, and **candS** is indexed on the pair (**cit, trieref**).

The Algorithm. The ATLaS implementation of Apriori is shown in Algorithm 4. First, we scan the dataset to find out frequent 1-itemsets and insert them into the trie. Next, we self-join the frequent 1-itemsets to generate candidate 2-itemsets. The WHERE condition on line 7 guarantees that each frequent itemset is uniquely represented in the trie — a child node is always labelled with a larger item than its parent. After the join (assuming we are mining the sample dataset in (6.1) with a threshold $MinSup = 2$), the contents of table **trie** and **cand** can be depicted by Figure 1(b). Finally, we invoke UDA **countset** to extend the trie to higher levels.

The implementation of **countset** is shown in Algo-



(a) Trie: each node represents a frequent itemset



(b) Trie and candidate itemsets on the 2nd-level

Figure 1: Representing the trie in a relational table with the reference data type

gorithm 5, which recursively extends the trie level by level until no more frequent itemsets can be found.

The INITIALIZE and ITERATE routine of UDA **countset** is responsible for counting the occurrences of each candidate. As we scan through each item in a transaction, we traverse the trie and incrementally find all the itemsets that are supported by the transaction, and we store the references to these itemsets in the **previous** table (line 7), which is initialized to contain nothing but the root node at the beginning of each transaction. On line 8, the count of the candidate is increased by 1 if the candidate itemset is supported by the transaction. We will now continue with our example starting from the trie in Figure 1(b): after the first transaction, (2, 3, 4), is processed by **countset**, table **previous** contains 4 nodes, namely the root, and nodes 2, 3, and 4; also, three candidate itemsets, (2, 3), (2, 4), and (3, 4), have their counts updated.

The TERMINATE routine of **countset** is responsible for extending the trie to a new level. On line 11, we call **nextlevel** to extend the trie to level J by adding candidates with a support no less than $MinSup$ to the trie. The UDA **nextlevel** also generates candidates on level $J + 1$. Then, we apply the anti-monotonic property to filter the candidates. This is achieved by calling **checkset** and **antimon** on line 12. Finally, on line 14, we recursively invoke **countset** to extend the trie to level $J + 1$ unless no new candidates are found.

The UDA **nextlevel** adds each qualified candidate onto the trie (line 5 in Algorithm 6). It also generates the next-level candidates by computing the self-join of the newly added itemsets; this UDA is called with a GROUP BY clause

Algorithm 5 Aggregate countset

```
1: AGGREGATE countset (bitem Int, J Int, MinSup Int,
   cans TABLE)
2: { TABLE previous(marked REF(trie), Level Int)
   INDEX(marked) MEMORY;
3: TABLE nextcands(cit Int, trieref REF(trie), freqcount Int)
   INDEX(trieref) MEMORY;
4: INITIALIZE: ITERATE: {
   /* Intialize previous for a new transaction if bitem=0. */
5: DELETE FROM previous WHERE bitem=0;
6: INSERT INTO previous VALUES (null, 0)
   WHERE bitem=0;
   /* Store supported frequent itemsets in previous */
7: INSERT INTO previous
   SELECT t.OID, p.Level+1
   FROM previous AS p, trie AS t
   WHERE t.itno=bitem AND t.father=p.marked
   AND p.Level<J-1;
   /* Count candidates that appear in the transaction */
8: UPDATE cans SET freqcount=freqcount+1
   WHERE bitem > 0 AND c.cit=bitem AND
   OID = (SELECT c.OID
   FROM previous AS p, cans AS c
   WHERE p.Level=J-1 AND c.trieref=p.marked);
9: }
10: TERMINATE: {
   /* Derive trie on level J and candidates on level J+1 */
11: INSERT INTO nextcands
   SELECT nextlevel (cit, trieref) FROM cans
   WHERE freqcount ≥ MinSup
   GROUP BY trieref;
   /* Eliminate candidates by the anti-monotonicity */
12: INSERT INTO subitems VALUES(null,0);
13: SELECT checkset(cit, trieref), antimon(cit, trieref, J)
   FROM nextcands;
   /* Ascend to level J+1 if cans not empty */
14: SELECT countset (b.item, J+1, MinSup, nextcands)
   FROM (SELECT count(*) AS size FROM nextcands)
   AS c, baskets AS b
   WHERE c.size > 0;
15: }
```

to exclude candidates that do not share the same parent³. The join operation is carried out through the use of a temporary table called **previous**, which stores all the itemsets that appear ahead of the current itemset, and they are joined with the current itemset to generate candidates on the new level. Figure 2(a) shows the result after **nextlevel** is applied: qualified candidates in Figure 1(b) become a new level of nodes in the trie, and a new set of candidates are derived by self-joining the itemsets on Level 2.

UDA **checkset** and **antimon** together implement the anti-monotonic property for pruning. For each candidate itemset on level $J + 1$, **checkset** traverses the trie to find all of its

³A candidate resulting from self-joining itemsets that do not share the same parent is already included in the join result of itemsets that share the same parent, or will be eliminated by the anti-monotonic property.

Algorithm 6 Supporting UDAs for Apriori

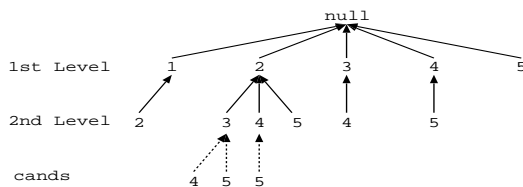
```
1: TABLE subitems(toid REF(trie), level Int) MEMORY;
   /* extend the trie and return candidates on the new level */
2: AGGREGATE nextlevel(item Int, ptrie REF(trie))
   : (Int, REF(trie), Int)
3: { TABLE previous(poid REF(trie)) MEMORY;
4: INITIALIZE: ITERATE: {
5: INSERT INTO trie VALUES(item, ptrie);
   /* join with previously inserted itemsets and */
   /* return them as next-level candidates */
6: INSERT INTO RETURN
   SELECT item, previous.poid, 0 FROM previous;
   /* appending the newly-added to the previous table */
7: INSERT INTO previous
   SELECT trie.OID FROM trie
   WHERE trie.itno=item AND trie.father=ptrie;
8: }
   /* for each (J+1)-itemset, find its frequent subsets of size J */
9: AGGREGATE checkset (citem Int, cref REF(trie))
10: { INITIALIZE: ITERATE: {
   /* call checkset recursively */
11: SELECT checkset(f.itno, f.father) FROM trie AS f
   WHERE cref<>null AND f.OID=cref;
   /* as we exit the recursion we expand subitems */
12: INSERT INTO subitems SELECT t.OID, s.level+1
   FROM subitems AS s, trie AS t
   WHERE t.itno=citem AND t.father=s.toid;
13: }
   /* pruning using the anti-monotonic property */
14: AGGREGATE antimon(it Int, aref REF(trie), J Int)
15: { INITIALIZE: ITERATE: {
16: DELETE FROM cans
   WHERE cans.cit=it AND trieref=aref
   AND J+1 > (SELECT count(*) FROM subitems
   WHERE subitems.level=J);
17: DELETE FROM subitems WHERE toid <> null;
18: }
```

sub-itemsets. According to the anti-monotonic property, a necessary condition for a $(J + 1)$ -itemset to be a frequent itemset is that each of its $J + 1$ subsets is a frequent itemset. Thus, **antimon** eliminates those candidates that have fewer than $J + 1$ frequent itemsets of size J . Figure 2(b) shows the result after **antimon** has eliminated candidate $(2, 3, 5)$ from Figure 2(a): $(2, 3, 5)$ cannot be a frequent itemset because one of its subset, $(3, 5)$, is not frequent.

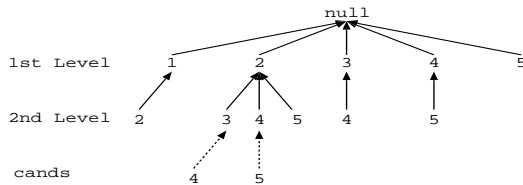
As shown in Figure 1(a), the process on the sample dataset terminates at level 3. At that point, table **trie** contain all the results, i.e., the frequent itemsets.

7 Performance

Our Apriori example shows that rather complex algorithms and data structures can be succinctly expressed in a very modest extension of SQL. To test the efficiency of this approach, we compared the performance of the Apriori



(a) After invoking UDA **nextlevel**



(b) After invoking UDA **checkset** and **antimon**

Figure 2: Candidates generation and pruning

algorithm implemented in ATLaS against the same algorithm implemented in C.

By default, ATLaS uses Berkeley DB as its record manager. The algorithm we compared with works directly on files in the file system, and we found that a sequential scan of a file is at least an order of magnitude faster than a sequential scan of a Berkeley DB table of the same content. In order to make a fair comparison, we use ATLaS table functions to access the data sets in the file system directly. That is, instead of using:

```
SELECT countset(item, 2, MinSup)
FROM baskets;
```

we use a user-defined table function **readFile**, which is implemented in external languages such as C, to read the basket data:

```
SELECT countset(b.item, 2, MinSup)
FROM TABLE (readFile('baskets')) AS b;
```

We tested the performance on synthetic basket data sets with millions of records; we produced data sets for the same sizes and by the same method as described by Agrawal et al. in [2].

We carry out our tests on a 766 MHz Pentium III running Linux OS 2.2.1 with 256M memory. Figure 3 shows the

Name	T	I	D	size of dataset
T5.I2.D100K	5	2	100K	2.8M text stream
T10.I2.D100K	10	2	100K	5.2M text stream
T10.I4.D100K	10	4	100K	5.2M text stream
T20.I2.D100K	20	2	100K	10.1M text stream

Table 3: Benchmark Data Sets

performance obtained on the four data sets shown in Table 3. Here, T is the average transaction size, I the average size of maximal large itemsets, and D the number of transactions. Using ATLaS instead of C causes a slowdown of about 50% for the smaller data sets, and of 30% for the larger ones. Thus the implementation of Apriori in ATLaS scales up at least as well as that in C. Similar performance overhead was obtained for classifiers. It suggests that, while there are still many opportunities for optimization in ATLaS, the performance overhead is modest; comparable slowdowns are frequently encountered when moving from one DBMS to another, and more severe slowdowns could be expected if we code in Java or Perl. Finally, the poor performance obtained (both in C and ATLaS) when storing **baskets** using the Berkeley DB storage manager, rather than a plain file, confirms that cache mining [18] remains the approach of choice in supporting data mining even when using an SQL-based language such as ATLaS.

We can compare the performance of the ATLaS approach vis a vis the results obtained in [18] where several alternatives were discussed: *C*, *Stored-Procedure*, *UDFs*, and *SQL-OR*. The *SQL-OR* approach embeds UDFs developed in C/C++ in SQL statements. Since the efficiency of the join algorithm has the most significant impact on the overall performance, in [18] S. Sarawagi et al. explored 6 different approaches, each implemented as carefully designed combinations of UDFs and SQL statements. Their experiments show that the *SQL-OR* approach based on vertical joins implemented in C/C++ to be the most efficient approach (for dataset T5.I2.D100K and DB2). The running time of the UDF and Stored Procedure approaches are respectively 1.5 and 2.2 times those of the O-R approach. We expect that the performance of ATLaS stored on DB2 falls between those of *SQL-OR* and UDF.

The performance of ATLaS on simpler applications that do not require special in-memory data structures is close to those of C/C++ programs. For instance a slow down of 15% was measured for the rollups using Algorithm 2. Now, current O-R systems support recursive queries and ROLAP extensions through built-in optimization procedures that are effective for typical cases. However, more efficient techniques are available for many special cases. Given the good performance and simplicity of programming provided by ATLaS, the user might prefer an ATLaS-based implementation of these efficient techniques, rather than the standard implementation provided by the system.

The significance of ATLaS having met the APriori performance test proposed in [2] should not be underestimated. Previous approaches, including our AXL system failed this test [22]. Indeed, AXL supported UDAs, but not table functions and in-memory tables, and various optimization techniques added to ATLaS to achieve the desired level of performance.

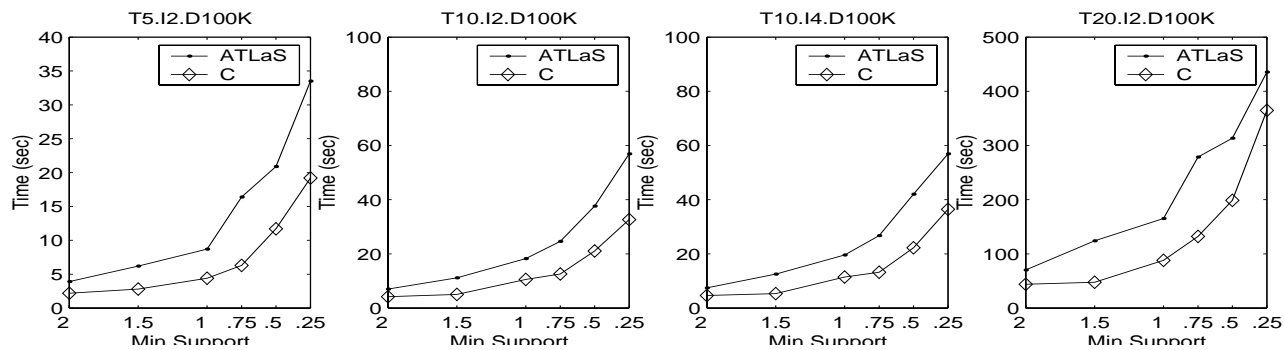


Figure 3: ATLaS vs. C implementation of Apriori (X axis is percentage of minimal support)

8 The ATLaS System

The ATLaS system has two main components: (i) the database storage manager, and (ii) the language processor.

The database storage manager consists of (i) the Berkeley DB library [20] and of (ii) additional access methods that we have implemented specifically for ATLaS. We use Berkeley DB to support access methods such as the B+Tree, and Extended Linear Hashing on disk-resident data. The additional access methods that we have implemented include R-trees (for secondary storage) and in-memory database tables with hash-based indexing. R-trees were introduced to support spatio-temporal queries, and in-memory tables were introduced to support the efficient implementation of special data structures, such as tries or priority queues, that are needed to support efficiently specialized algorithms, such as Apriori or greedy graph-optimization algorithms.

The ATLaS language processor translates ATLaS programs into C++ code, which is then compiled and linked with the database storage manager and user-defined external functions. For instance, the 60 or so lines in the Apriori algorithm presented in Section 5 compile into more than 2,800 lines of C++ code. The current version of ATLaS (Version 2.2) works under UNIX as well as MS Windows [1].

The implementation of the ATLaS system can be broken down into three major modules: the parser, the rewriter and the code generator (Figure 4). The core data structure

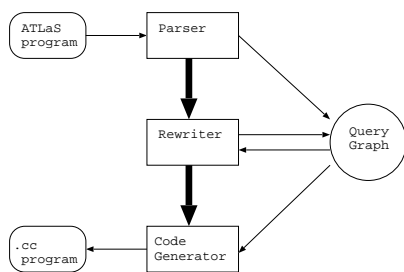


Figure 4: The ATLaS Architecture

is the query graph, which is used by all ATLaS modules. The parser builds initial query graphs based on the abstract syntax tree. The rewriter makes changes to the query graphs. Finally, the code generator translates the query graphs into C++ code.

A query graph is composed of a set of nodes and arcs. A node represents some kind of operation, for instance, SELECT, INSERT and DELETE. Some operations can be pipelined (SELECT statement without the DISTINCT flag), others are blocking (ORDER-BY, GROUP-BY, and etc.). An arc connecting a parent node and a child node indicates that the parent node consumes the data stream produced by the child node.

After initial query graphs are built during parsing, the rewriter makes changes to the query graphs. The rewriter is a very important module, since much optimization, such as predicate push-up/push-down, UDA optimization, index selection, and in-memory table optimization, is carried out during this step. While ATLaS performs sophisticated local query optimization, it does not attempt to perform major changes in the overall execution plan, which therefore remains under programmer's control. More generally, system-controlled schemes for optimizing and parallelizing UDAs provide an interesting topic for further research.

The runtime model of ATLaS is based on data pipelining. In particular, all UDAs, including recursive UDAs that call themselves, are pipelined, which means tuples inserted into the RETURN relation during the INITIALIZE/ITERATE steps are sent to their caller immediately after the UDA finishes processing the current item, instead of after it finishes processing the entire stream. Therefore, local variables (temporary tables) declared in a UDA can not reside on the stack; these variables form the current state of the aggregation and they are needed when the UDA resumes processing of the next item. We assemble these variables into a **state** structure which is then passed to the UDA for each INITIALIZE/ITERATE/TERMINATE call, so that these internal data are retained between calls.

The current ATLaS implementation has 42,000 lines of

C++ code. It is robust enough to be downloaded and used for course work [1], although there remains plenty of room for improvements—e.g., several SQL data types are not yet supported, and will be added in the future.

9 Conclusion

With ATLaS, we have proposed and demonstrated minimal extensions that turn SQL into a powerful database language, which is conducive to the development of Database-centric data mining applications. The challenges posed by this problem is illustrated by the approaches previously proposed [15, 6, 9, 18]; in particular, a study presented in [18] established APriori as a task clearly beyond the capability of current DBMS technology. In this paper, we have shown how ATLaS satisfies this difficult test.

While the notion of extensible databases has been discussed by researchers for a long time, the approach of enhancing SQL with native extensibility mechanisms has not been studied in the past (at the best of our knowledge). Vendors have approached the problem by either providing narrow extensions for specific applications, or by exploding SQL into a full programming language, such as PL/SQL or SQL/PSM. Here instead, we have taken a minimalist's approach, and determined what extensions are required for turning SQL into a language supportive of data mining. We have shown that SQL-coded UDAs and table functions, combined with in-memory tables and various optimization techniques, allow us to express data mining applications concisely and efficiently. In the paper we have discussed decision tree classifiers and the APriori algorithm. Other data mining applications, including clustering and time series analysis, are discussed in [1]. Some spatio-temporal data mining applications are discussed in [4].

Acknowledgments This research was supported in part by NSF grant IIS 0070135.

References

- [1] ATLaS download page. <http://wis.cs.ucla.edu/atlas>.
- [2] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proc. of Very Large Database (VLDB)*, 1994.
- [3] F. Bonchi, F. Giannotti, G. Mainetto, and D. Pedreschi. A classification-based methodology for planning audit strategies in fraud detection. In *Int'l Conf. on Knowledge Discovery and Data Mining (SIGKDD)*, San Diego, CA, August 1999.
- [4] Cindy Chen, Haixun Wang, and C. Zaniolo. An extensible database system for spatio-temporal queries. Submitted for publication, 2003.
- [5] F. Giannotti, G. Manco, D. Pedreschi, and F. Turini. Experiences with a logic-based knowledge discovery support environment. In *ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery (DMKD)*, 1999.
- [6] J. Han, Y. Fu, W. Wang, K. Koperski, and O. R. Zaiane. DMQL: A data mining query language for relational databases. In *Workshop on Research Issues on Data Mining and Knowledge Discovery (DMKD)*, pages 27–33, Montreal, Canada, June 1996.
- [7] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *Int'l Conf. Management of Data (SIGMOD)*, 2000.
- [8] J. M. Hellerstein, P. J. Hass, and H. J. Wang. Online aggregation. In *Int'l Conf. Management of Data (SIGMOD)*, 1997.
- [9] T. Imielinski and A. Virmani. MSQL: a query language for database mining. *Data Mining and Knowledge Discovery*, 3:373–408, 1999.
- [10] Informix. Informix: Datablade developers kid infoshelf. <http://www.informix.co.za/answers/english/docs/dbdk/infoshelf>, 1998.
- [11] ISO DBL LHR-004 and ANSI X3H2-95-364. (ISO/ANSI working draft) database language SQL3, 1995.
- [12] ISO/IEC JTC1/SC21 N10489, ISO/IEC 9075. Committee draft (cd), database language SQL, July 1996.
- [13] T. Johnson, Laks V. S. Lakshmanan, and Raymond T. Ng. The 3w model and algebra for unified data mining. In *Proc. of Very Large Database (VLDB)*, pages 21–32, 2000.
- [14] Yan-Nei Law, Haixun Wang, and Carlo Zaniolo. Blocking, monotonicity, and turing completeness in a database language for sequences and streams. <http://wis.cs.ucla.edu/publications/turing.pdf>.
- [15] R. Meo, G. Psaila, and S. Ceri. A new SQL-like operator for mining association rules. In *Proc. of Very Large Database (VLDB)*, pages 122–133, Bombay, India, 1996.
- [16] K. Rajamani, A. L. Cox, B. R. Iyer, and A. Chadha. Efficient mining for association rules with relational database systems. In *International Database Engineering and Applications Symposium (IDEAS)*, pages 148–155, 1999.
- [17] Reza Sadri, Carlo Zaniolo, and Amir M. Zarkesh and Jafar Adibi. A sequential pattern query language for supporting instant data mining for e-services. In *Proc. of Very Large Database (VLDB)*, pages 653–656, 2001.
- [18] S. Sarawagi, S. Thomas, and R. Agrawal. Integrating association rule mining with relational database systems: Alternatives and implications. In *Int'l Conf. Management of Data (SIGMOD)*, 1998.
- [19] C. Shafer, R. Agrawal, and M. Mehta. SPRINT: A scalable parallel classifier for data mining. In *Proc. of Very Large Database (VLDB)*, 1996.
- [20] Sleepycat Software, <http://www.sleepycat.com>. *The Berkeley Database (Berkeley DB)*.
- [21] Haixun Wang and Carlo Zaniolo. Nonmonotonic reasoning in $\mathcal{LDL}++$. In Jack Minker, editor, *Logic-Based Artificial Intelligence*, pages 523–542. Kluwer Academic Publishers, 2000.
- [22] Haixun Wang and Carlo Zaniolo. Using SQL to build new aggregates and extenders for object-relational systems. In *Proc. of Very Large Database (VLDB)*, 2000.
- [23] Haixun Wang, Carlo Zaniolo, and Chang Luo. ATLaS: a turing-complete extension of sql for data mining applications and streams. <http://wis.cs.ucla.edu/atlas/doc/>.