The ATLaS User Manual

Haixun Wang
Carlo Zaniolo
Richard Luo

April 22, 2004

Contents

Chapter 1

Introduction

ATLaS is an SQL-based programming language for data-intensive applications. Unlike languages, such as PL/SQL or SQL/PSM, which use the imperative the constructs of procedural languages, ATLaS achieves Turing completeness by using declarations, i.e., by supporting the declarations of new aggregates and table functions.

An ATLaS program consists of list of declarations followed by a list of SQL statements. Therefore, (with `A|B` denoting either `A` or `B`, and `A*` denoting zero or more occurrence of `A`) we have:

ATLaS-program	→	ATLaS-dcl∗ SQL-statement∗
ATLaS-dcl	→	Table-dcl | UDA-dcl | TFunc-dcl

Figure 1.1: Syntax of ATLaS extensions

The SQL-statements mentioned in Figure 1.1 are the basic select, insert, delete, update commands of SQL-2, which are summarized in Figure 2.2. We will assume that our reader is already familiar with SQL-2 and concentrate on the extensions which make makes ATLaS so powerful. ATLaS is Turing-complete because of its declarations, which are of three kinds:

1. Table declarations (Table-dcl]),
2. declarations of User-Defined Aggregates (UDA-dcl),
3. declarations of Table Functions (TFunc-dcl).

In the next Section we discuss simple programs that only use table declarations. Programs containing user-defined aggregates and table functions are discussed in the following sections.

Chapter 2

ATLaS SQL on Tables

ATLaS can use a variety of tables from different sources. In particular, it can access B+Tree indexed tables managed by the embedded database system Berkeley DB [5]. For instance, say that an employee table, with key `Eno`, is stored in such a format in the directory
`C:\mydb\employees`. [Say more about win vs linux] Then the following program first gives a 5 % raise to the employees in the 'QA' department, and then prints all the employees who now make more than 60K:

```
                    /* Begin of ATlaS program— this is a comment*/
table employees(Eno int, Name char(18), Sal real, Dept char(6))
          Btree(Eno)   source  'C:\mydb\employees';
update employees set Sal = Sal * 1.05
       where Dept='QA' ;
insert into stdout select Eno, Name
       from employees where Sal > 60000;
                    /* End of ATlaS program */
```

The `insert into stdout` clause before `select` in the last statement can be omitted without changing the meaning of this program. Thus ATLaS supports the standard `select, insert, delete, update` statements of SQL-2. Observe that keywords can be written in upper case or lower case—however, attribute names and other user-defined identifiers are case-sensitive.

Since we assume that our readers are already familiar with SQL-2, we will now concentrate on the new constructs of ATLaS and return to SQL in Section 6.

Three types of tables are currently supported in ATLaS, as follows:

1. Secondary storage tables with indexed by B+ trees on one or more attributes, as in the following example:

   ```
   TABLE employees(Eno int, Name char(18), Sal real, Dept char(6))
         BTree(Eno) source 'C:\mydb\employees';
   ```

   The `SOURCE` declaration associates this table with the file `C:\mydb\employees` and make it *persistent*. Persistent tables remain after the the program completes its

execution. Tables declared without a `source` are *transitory* and they are removed at the end of the program execution.

2. Secondary storage tables with indexed by R+ trees on a pair or a quadruplet of real attributes: the first can be used to index points, and the second for rectangles:

```
TABLE mypoints(x real, y real, object char(10)) RTree(x,y);

TABLE myrectangles(tx real, ty real, bx real, by real, object char(10))
                                        RTree(tx,ty,bx,by);
```

3. Main memory tables, with a hash-based index on one or more attributes:

```
TABLE memo(j Int, Region Char(20))
          MEMORY AS VALUES(0, 'root-of-tree');
```

   This example also illustrates that a transitory table can be initialized to the results of the query defined using the `AS query` option. In this example, we use constant values to initialize the table. In general, the initialization query can use the content of previously declared tables.

A source declaration can only be given for B+tree and R+tree tables, but not for `MEMORY` tables since these are never persistent. The syntax of table declarations is as follows:

Table-dcl	→	'TABLE' table-id '(' columns keydec ')'
		['SOURCE' | ''file-name'' 'AS' query] ';'
column-list	→	column [',' column]*
column	→	id type
type	→	'INT' | 'REAL' | 'CHAR' '(' num ')' | 'REF' '(' id ')'
keydec	→	['BTree('key')', 'RTree('key')', MEMORY] ';'
key	→	'(' id [',' id]* ')'
load	→	'LOAD' 'FROM' ''file-name'' 'INTO' table-id

Table 2.1: Declaring and Initializing Tables in ATLaS.

The `LOAD` construct of ATLaS can be used to load into a table data from an external file, where the commas (carriage returns) are used as separators between attribute values (records). Newly loaded tuples are appended to the existing tuples.

SQL Statements

SQL-statement	→	select-st| delete-st| insert-st| update-st
select-st	→	query [order-clause] ';'
order-clause	→	'ORDER' 'BY' exp ['ASC' | 'DSC'] [',' exp ['ASC' | 'DSC']]*
query	→	query-block [set-op query-block]*
set-op	→	'UNION'|'INTERSECT'|'EXCEPT'
query-block	→	'SELECT' hxp [, hxp]* 'FROM' squn [',' squn]* ['WHERE' exp] ['GROUP' 'BY' exp [',' exp]* ['HAVING' exp]
delete	→	'DELETE' 'FROM' id ['WHERE' exp] ';'
insert	→	'INSERT' 'INTO' id select-st ';' VALUE ... not defined?
update	→	'UPDATE' id 'SET' update-list ['WHERE' exp] ';'
update-list	→	id '=' exp [',' id '=' exp]*
exp	→	'NIL' | num | float | string | id ['.' id] | ref | exp ('='|' < '|' <='|'>'|'>= '|'<>') exp | exp ('AND'|'OR'|'IN'|'NOT' 'IN') exp | 'EXISTS' exp | ('max'|'min'|'count'|'sum'|'avg') '('exp')' | '(' exp ')' | '(' query ')'
ref	→	ref '→' id | id ['.' id] '→' id
id	→	letter [letter | digit]*
letter	→	('a'-'z'|'A'-'Z')
digit	→	('0'-'9')
hxp	→	exp [['AS'] hxp-alias] | [id '.'] '*'
hxp-alias	→	id | '(' id [',' id]* ')'
qun-alias	→	['AS'] id ['(' id @ ',' id ')']

Table 2.2: Syntax of the basic SQL Statements supported in ATLaS

Chapter 3

User-Defined Aggregates

ATLaS supports the standard five aggregates `count, sum, avg, min`, and `max` without the `DISTINCT` option. But the real power of ATLaS follows from its User-Defined Aggregates (UDAs) discussed next. As a first example, we define an aggregate equivalent to the standard `avg` aggregate in SQL.

Standard Average 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`. To compute the average, the SQL statement in `INITIALIZE` inserts the value taken from the input stream and sets the count to 1. 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):

```
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;
   }
}
```

The basic initialize-iterate-terminate template used to define the average aggregate of SQL-2, can now be used to defined powerful new aggregates required by new database applications.

OnLine Average For instance, there is much current interest in online aggregates [2]. Since averages converge toward the correct 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.) Online averages can be expressed in ATLaS as follows:

```
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 : {  }
}
```

Therefore, the online average program has been obtained from the traditional average program by removing the statements from `TERMINATE` and adding a `RETURN` statement to `ITERATE`. Our UDA `online_avg` takes a stream of values as input and returns a stream of values as output (one every 200 tuples). In this example only one tuple is added to output by the the INSER INTO RETURN statement; in general, however, such statement can produce (a stream of) several tuples. Thus ATLaS UDAs operate as general stream transformers.

ATLaS uses the same basic framework to define both traditional aggregates and non-blocking aggregates. ATLaS UDAs are non-blocking when their `TERMINATE` clause is either empty or absent.

The typical default semantics for SQL aggregates is that the data is first sorted according to the `GROUP-BY` attributes; this 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.

Calling User-Defined Aggregates (UDAs) 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.

SQLCODE This a convenient labor-saving device found in most SQL systems, that comes very handy for the ATLaS programmer who wants to correlate a statement with the next. SQLCODE is set to a positive value when the last statement had a null effect, and to zero otherwise. Thus to tell the user that no employee was found in department 1024, we can modify the previous program as follows:

```
SELECT Sex, online_avg(Sal)
FROM employee WHERE Dept=1024 GROUP BY Sex;
select 'Nobody found in that department'
        where SQLCODE >0;
```

In the last statement, the predicates in the `WHERE` clause controls its conditional execution, in a fashion similar to that of the `IF` clauses in a procedural programming language. In fact, the ATLaS compiler recognizes, and optimizes execution of, such conditional predicates.

Minima: Points and Values. In the next Example, we have a sequence of point-value pairs, and we define a `minpair` aggregate that returns the point where a minimum occurs along with its value at the minimum.

```
AGGREGATE minpair(iPoint Int, iValue Int):  (mPoint Int, mValue Int)
{   TABLE mvalue(value Int) MEMORY; TABLE mpoints(point Int) MEMORY;
    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 have used two internal tables: the `mvalue` table holds, as its only entry, the current min value, while `points` holds all the points where this value occurs. In the `ITERATE` statement we have used `SQLCODE` to 'remember' if the previous statement updated `mvalue`; this is the situation in which the old value was larger than the new one and the old points must be discarded.

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.

For instance, say that we have a time series containing the daily closing prices of certain stocks arranged in temporal sequence (i.e. the table `stock_prices`, below). Then the following program computes the local minima for each stock:

```
/* The declaration of AGGREGATE minpair should go here*/
TABLE stock_prices(Day Int, Stock char(4), Cprice Real)
                                    source 'D:\mydabase\stock_prices'
  select Stock, minpair(Day, Stock) → iPoint, minpair(Day, Stock)→ iValue
  from stock_prices
  group by Stock ordered by Stock, minpair(Day, Stock) → iPoint
```

Observe the use of "→" to identify the different components in the two-column tuples returned by the aggregate `minpair`. Since temporal data types are not yet supported in the current ATLaS version, we are using integers to represent dates: thus May, 27, 1999 is represented as `19990527`.

The next table summarizes the syntax for declaring new aggregates.

UDA-dcl	→	'AGGREGATE' id '(' column-list ')' ':' return-type aggr-body
aggr-body	→	'{' ATLaS-dcl*
		'INITIALIZE' ':' { SQL-statement* ';'}'
		'ITERATE' ':' { SQL-statement* ';' '}'
		'TERMINATE' ':' { SQL-statement* ';'}'
		'}'
return-type	→	type | '(' column-list ')'

Figure 3.1: The declaration of UDAs

Initializing Tables and Combining Blocks. Let us now introduce the following two syntactic variations of convenience supported in ATLaS:

- Tables declared in UDAs can be initialized as part of their declaration, via an SQL statement. This executes at the time when the first tuple is processed, thus the result is the same as if the initialization had been executed in the INITIALIZE block.
- Different blocks can be merged together when they perform the same function. In the next example the INITIALIZE and ITERATE blocks are merged together.

Our Online Averages UDA could also have been written as follows:

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

In the previous example, the statement has been omitted: this is equivalent to writing '`TERMINATE: { } `'. An empty `INITIALIZE` statement can also be omitted in a similar fashion.

The results produced by online averages depend on the order in which the data is streamed through the UDA. This illustrate a common situation in stream processing: the abstract semantics of the aggregate used is order-independent, but approximations must be used because of efficiency and real-time requirements (e.g., nonblocking computations); often, the approximate UDA is order-dependent.

In other situations, no approximation is involved, and the dependence on order follows from the very semantics of the UDA. For instance, this is the case of the `rising aggregate` described below.

Rising. In addition to temporal extensions of standard aggregates (suggested homework: write them in ATLaS), TSQL2 [7] proposes this new aggregate to return the maximal time periods during which a certain attribute values has been increasing monotonically. We can apply this aggregate to our stock_prices(Day Int, Stock char(4), Cprice Real) table to find the periods during which different stocks have been rising, as follows:

```
select Stock, rising(Day, Cprice) → Start, rising(Day, Cprice)→ End
from stock_prices
group by Stock
```

where `rising` is defined as follows:

```
AGGREGATE rising(iPoint Int, iValue Real) : (Start Int, End Int)
{ TABLE rperiod(First Int, Last Int, Value Real) MEMORY;
   INITIALIZE: {
         INSERT INTO rperiod VALUES (iPoint, iPoint, iValue);
      }
   ITERATE: {
         INSERT INTO return SELECT First, Last
              FROM rperiod
              WHERE iValue <= Value AND First < Last;
         UPDATE rperiod SET Last=iPoint, Value=iValue
              WHERE iValue > Value;
         UPDATE rperiod SET First=iPoint, Last=iPoint, Value=iValue
              WHERE SQLCODE > 0;
      }
   TERMINATE:{ INSERT INTO return SELECT First, Last
              FROM rperiod
              WHERE First < Last; }
      }
}
```

Therefore we have a sequence of time-value ordered by increasing time. We store a zero length period `iPoint, iPoint` whenever the new `iValue` is not increasing (also at `INITIALIZE`). Also a non-zero length period is currently held in `rperiod` we return it. When the new `iValue` is larger than the previous stored value, we advance the `End` of the current period to the current point.

Chapter 4

Table Functions

Table functions play a critical role in rearranging data, and allowing the composition of aggregates. For instance, if we want to count the number of the local minima found in the execution of `minpair`, we can use the following table function:

Cascading of Aggregates via a Tfunction

```
/* include the declaration of the AGGREGATE minpair here*/
TABLE stock_prices(Day Int, Stock char(4), Cprice Real)
                              source 'C:\mydabase\stock_prices' ;
FUNCTION localmins():(Stock Int, Point Int, Value Int)
   {  insert into RETURN
      select p.Stock, minpair(p.Day, p.Cprice)
         from stock_prices as p
         group by p.Stock
      }
select L.Stock, count(L.Point)
   from stock_prices, TABLE(localmins()) AS L
   group by L.Stock;
```

Here the table function does little more than calling the `minpair` aggregate on the `stock_prices` table, and organizing the results as a virtual table with attributes `(Stock, Point, Value)`. Then, the aggregate `count` can be called on this virtual table, whereas the direct cascading of aggregates is not allowed in SQL, nor in ATLaS.

Also observe that the notation `TABLE( ...)` must be used when invoking table functions, to conform to SQL standards.

The final NB is that the 'dot' notation (e.g., `L.Point`) is used to refer to the various columns in a tuple produced by a table function, whereas we use"$\rightarrow$" for UDAs (e.g., `minpair(Day, Stock)` $\rightarrow$ `iPoint`).

Pre-Sorting the Data The correctness of the `rising` is predicated upon the data being sorted by increasing time. If that is not the case, we can use a table function to pre-sort the data. Then, our program becomes:

```
/*AGGREGATE minpair include here the rising declaration*/
TABLE stock_prices(Day Int, Stock char(4), Cprice Real)
                          source 'C:\mydabase\stock_prices' ;
FUNCTION sort():(Stock Int, Point Int, Value Int)
   {  insert into RETURN
      select Day, Stock, Cprice)
         from stock_prices
         ORDERED BY Stock, Day
      }
   insert into stdout
   select Stock, rising(Day, Stock) → Start, rising(Day, Stock)→ End
   from stock_prices, TABLE(sort())
   group by Stock
```

Column Value Pair The first step for most scalable classifiers is to convert the training set into column/value pairs. For instance, say that our training set is as follows, where the various conditions are described by their initials (e.g., S, O, R in the first column stand respectively for Sunny, Overcast, and Rain):

RID	Outlook	Temp	Humidity	Wind	Play
1	S	H	H	W	N
2	S	H	L	S	N
3	O	H	L	W	Y
4	R	M	H	W	Y
...	...	...	...	...	...

Figure 4.1: The relation **PlayTennis**

Then, 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`. For instance, the first tuple should produce the following tuples:

$$(1,\mathtt{S},\mathtt{N}),\ (2,\mathtt{H},\mathtt{N}),\ (3,\mathtt{H},\mathtt{N}),\ (4,\mathtt{W},\mathtt{N})$$

Our next table function, called `dissemble` can be used for the task.

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

The syntax for function tables is given below. The mapping expressed by `dissemble` could be expressed in standard SQL via the union of several statements, but such a formulation could lead to inefficient execution. In later sections we will show that,

Tfunc-dcl	→	'FUNCTION' id '(' column-list ')' ':' return-type Tfunc-body
return-type	→	type | '(' column-list ')'
Tfunc-body	→	'{' ATLaS-dcl* SQL-statement* '}'

Figure 4.2: The declaration of a Table Function

using this table function and specialized aggregates we can express Naive Bayesian classifiers and Decision Tree Classifiers in a few ATLaS statements.

Chapter 5

Programming in ATLaS

5.1 Recursion

Example 1 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 1 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.) Observe that the `INITIALIZE` and `ITERATE` routine in Example 1 share the same block of code.

Example 1 *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';
```

Besides the Prolog-like top-down computation of Example 1, we can also express easily the bottom-up computation used in Datalog, 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 Figure 5.1; 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 5.1.

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

Figure 5.1: The relation **PlayTennis**

The first step for most scalable classifiers [4] is to convert the training set into column/value pairs. This conversion, although conceptually simple, is hard to express succinctly in SQL. Consider the `PlayTennis` relation as shown in Figure 5.1. 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 2 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 2 *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 (**count(Value)$\leq$ 1**), or tuples in the partition belongs to one class (**sum(Yc)=0** or **sum(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` previously defined returns the minimal gini index as well as the column

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)
   BTree(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() recusively 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 AND
             t.col NOT IN (SELECT col FROM mincol)
       GROUP BY m.Value;
     }
15: }
```

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 Figure 5.1). The recursion terminates if table `mincol` is empty, that is, there is no valid column to further split the partition.

To classify the PlayTennis dataset shown in Table 5.1, we use the program:

```
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). The fact that a concise expression for this algorithm is now possible suggests the stream-oriented computation model of UDAs adds to the expressive power of ATLaS.

5.2 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 `UNION`s. For instance, to roll up a sales table along dimensions such as Time, Region, and Department, we can use the query of Example 3.

Example 3 *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 3, 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 a in-memory table to keep track of the subtotals at each rollup level j (j = 1, ..., 4, with level 4 corresponding to the grand total). The table is as follows:

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

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

```
 1: AGGEGATE 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=10 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=1;
      }
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: }
```

MEMORY

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 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

Algorithm 3 Sorting and then rolling-up 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: }
```

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>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.

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.

5.3 References and Data Structures

In ATLaS, the types of table columns and the types of parameters to User-Defined aggregates can be any of the following:

- **INT**
- **REAL**
- **CHAR(n)**
- **REF**

The reference type, denoted by REF, is currently supported only for *in-memory* tables, which are declared with the MEMORY option, as in following example:

```
TABLE faculty(name CHAR(20), dept CHAR(20)) MEMORY;
```

In-memory tables can have columns of Reference types, which are essentially points to tuples in some other tables (or refer to themselves). For example, we can define the following tables:

```
TABLE faculty(name CHAR(20), dept REF(department)) MEMORY;
TABLE department(name CHAR(20), chair REF(faculty)) MEMORY;
```

We can find out the name of the chair of the CS department by using the following query:

```
SELECT chair->name
FROM department
WHERE name = 'CS';
```

Object ID and Path Expression In ATLaS, each tuple in an in-memory table has its unique OID (object id). In SQL statements, we treat OID of a tuple as its pseudo column. The type of the OID column is `REF(table)`, where table is the table the tuple is in.

The following example demonstrates the use of OID and reference types.

```
TABLE tree(name char(10), father REF(tree)) MEMORY;

INSERT INTO tree
SELECT 'mary', t.OID
FROM tree AS t
WHERE t.name = 'tom';
```

In the above example, we define a table with a column that refers to a tuple in the same table. The subsequent INSERT statement creates a new tuple whose father is another tuple in the table with name 'tom'. The expression `t.OID` retrieves the OID of the current tuple.

With reference types and OID and we can use path expressions to navigate through the tables. The following query finds the name of Jane's grandfather. Note that if t is of reference type, then t and `t->OID` are the same thing, which means `father->name` is the same as `father->OID->name`.

```
SELECT father->father->name
FROM tree AS t
WHERE t.name = 'jane';
```

A critical application of reference types and path expression is the implementation of in-memory data structures that are critical for the performance of many algorithms, including the Apriori algorithm discussed next.

5.4 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 [3] also established the Apriori algorithm as the litmus test that any aspiring solution must satisfy. The AXL system [6] failed this acid test—also all the applications presented 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 [1]. Let $I = \{i_1, ..., i_m\}$ 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:

$$\mathbf{0}, 2, 3, 4, \mathbf{0}, 1, 2, 3, 4, \mathbf{0}, 3, 4, 5, \mathbf{0}, 1, 2, 5, \mathbf{0}, 2, 4, 5 \tag{5.1}$$

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 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 5.2(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 5.2(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

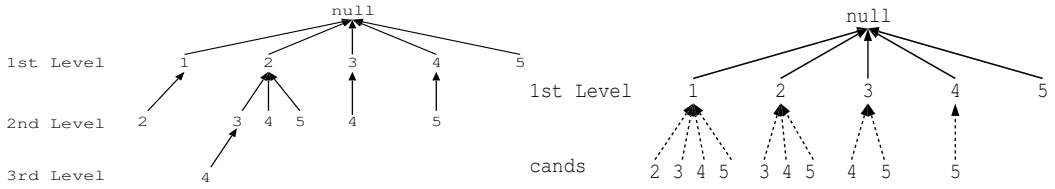
Algorithm 4 Main ATLaS Program for Apriori

```
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;
   /* intialize the trie to contain frequent one-itemsets */
6: INSERT INTO trie SELECT item, null FROM fitems;
   /* self-join frequent 1-itemsets to get 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. Start with k=2 */
8: SELECT countset(item, 2, MinSup, cands) FROM baskets;
```

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 operation (assuming we are mining the sample dataset in (5.1) with a threshold $MinSup = 2$), the contents of table `trie` and `cand` can be depicted by Figure 5.2(b). Finally, we invoke UDA `countset` to extend the trie to higher levels.

The implementation of `countset` is shown in Algorithm 5, which recursively extends the trie level-by-level until no more frequent itemsets can be found.



(a) Trie: each node represents a frequent itemset

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

Figure 5.2: Representing the trie in a relational table with the reference data type

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,

Algorithm 5 countset

```
1: AGGREGATE countset (bitem Int, J Int, MinSup Int, cands 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 cands SET freqcount=freqcount+1
         WHERE bitem > 0 AND c.cit=bitem
           AND OID = (SELECT c.OID FROM previous AS p, cands 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 cands 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 cands 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: }
```

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 5.2(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.

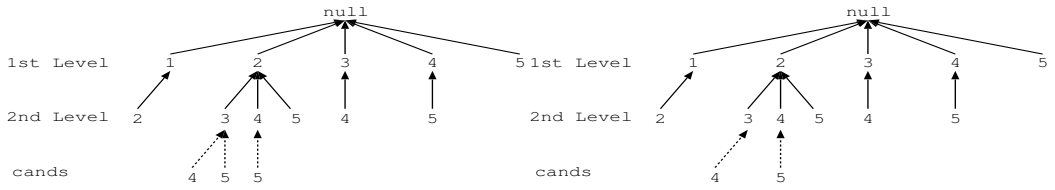
The `TERMINATE` routine of `countset` is responsible for extending the trie to a new level. On line 11, we call `nextlevel` to extend the trie to level J by adding candidates with a support no less than $MinSup$ to the trie. The UDA `nextlevel` also generates

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.

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 candi-
    dates */
 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 cands
          WHERE cands.cit=it AND trieref=aref
            AND J+1 > (SELECT count(*) FROM subitems WHERE
    subitems.level=J);
17:     DELETE FROM subitems WHERE toid <> null;
      }
18: }
```

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 to exclude candidates



(a) After invoking UDA **nextlevel**

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

Figure 5.3: Candidates generation and pruning

that do not share the same parent[1]. 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 5.3(a) shows the result after **nextlevel** is applied: qualified candidates in Figure 5.2(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 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 5.3(b) shows the result after **antimon** has eliminated candidate $(2,3,5)$ from Figure 5.3(a): $(2,3,5)$ cannot be a frequent itemset because one of its subset, $(3,5)$, is not frequent.

As shown in Figure 5.2(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.

[1]A 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.

Chapter 6

External Functions

ATLaS supports both scalar external functions and table external functions.
(For latest update, please refer to our website.)

6.1 Scalar Functions

To declare an external function to be dynamically loaded into ATLaS, we use the following syntax:

```
external int ginif(a int) in 'gini.so';
```

The above statement declares a UDF *ginif* which takes one integer-type parameter and returns an integer result. This function is supported by a shared library, 'gini.so'.

We can use C or any other language to create functions in shared libraries. On a UNIX system, the following command compiles C source code to dynamical libraries.

```
gcc -shared -o gini.so -fPIC gini.c
```

Once defined, the UDF can be used in ATLaS. For instance:

```
select gini(a) from test;
```

In order to dynamically load the library, the OS must be able to find it. In UNIX, the OS searchs for the library in all the paths specified by the environment variable LD_LIBRARY_PATH.

6.2 Table Functions

In much the same way, we can use external UDF as table functions.

For instance, we want to stream through the first K Fibonacci numbers. It is not difficult to write a C function to generate the Fibonacci numbers. The following ATLaS program demonstrates how to use such an external table function.

```
external table (i int, f int) fib(k int) in 'tabf.so';

select t.i, t.f
from table (fib(10)) as t;
```

In order to declare an external table function, we must use TABLE as the return type. The above declaration indicates 'fib' is an external function found in shared library 'tabf.so', and 'fib' returns a stream of tubples (i,f), where f is the i-th Fibonacci number. Then, in the following query, we stream through the first 10 Fibonacci numbers by calling 'table (fib(10))'.

How do we implement a table function in C? Unlike stateless scalar functions, table functions must keep their internal state between calls. More specifically, the function must be able to: i) determine the first call from subsequent calls; ii) tell the caller whether a tuple is successfully returned; iii) use a mechanism to return tuples to the caller. As an example, the following code implements function 'fib':

```
#include <db.h>

struct result {
  int a;
  int b;
};

int fib(int first_entry, struct result *tuple, int k)
{
  static int count;
  static int last;
  static int next;

  if (first_entry == 1) {
    count = 0;
    next=1;
    last=0;
  }
  if (count++ <k) {
    tuple->a = count;
    tuple->b = last;

    last = next;
    next = next+tuple->b;
    return 0;
  } else {
    return DB_NOTFOUND;
  }
}
```

In addition to the arguments (here is 'k') passed to the table function, we have 2 extra arguments: i) *first_entry*, if first_entry=1 then it is the first call; ii) tuple, which

is a pointer to a structure where results are to be stored. External table functions always return an integer value, 0 if successful, DB_NOTFOUND otherwise.

A possible use of table functions is to scan file system data, and return results to the database system after filtering. Our test indicates that on a linux system, external table functions accessing file system data is almost 100 times faster than accessing the same data in the Berkeley DB format.

6.3 Built-in Aggregates and Functions

ATLaS supports the standard builtin aggregates: **min()**, **max()**, **sum()**, **avg()**, and **count()**.

ATLaS supports the following builtin functions: (they are being added constantly.)

- **srand(INT) : INT**
  The srand() function sets its argument as the seed for a new sequence of pseudo-random integers to be returned by rand(). These sequences are repeatable by calling srand() with the same seed value. srand() always returns 0.

- **rand() : REAL**
  The rand() function returns a pseudo-random real between 0 and 1. The following code set 10 as a random seed, and displays two random values.

```
VALUES(srand(10));

VALUES(rand(), rand());
```

- **sqrt(REAL) : REAL**
  The sqrt(x) function returns the non-negative square root of x.

- **timeofday() : CHAR(20)**
  The gettimeofday function gets the system's notion of the current time. The current time is expressed in elapsed seconds and microseconds since 00:00 Universal Coordinated Time, January 1, 1970. It returns a string in the form of x'y", where x is the seconds and y is the microseconds. This function is maily used to measure the performance of ATLaS queries, as in the following example:

```
INSERT INTO stdout VALUES(timeofday());

... some ATLaS queries ...

INSERT INTO stdout VALUES(timeofday());
```

Bibliography

[1] R. Agrawal, R. Srikant. "Fast Algorithms for Mining Association Rules". In *VLDB'94*.

[2] J. M. Hellerstein, P. J. Haas, H. J. Wang. "Online Aggregation". *SIGMOD, 1997.*

[3] S. Sarawagi, S. Thomas, R. Agrawal, "Integrating Association Rule Mining with Relational Database Systems: Alternatives and Implications". In *SIGMOD, 1998.*

[4] J. C. Shafer, R. Agrawal, M. Mehta, "SPRINT: A Scalable Parallel Classifier for Data Mining," In *VLDB 1996.*

[5] Sleepycat Software, "The Berkeley Database (Berkeley DB)", http://www.sleepycat.com.

[6] H. Wang and C. Zaniolo: Using SQL to Build New Aggregates and Extenders for Object-Relational Systems. VLDB 2000.

[7] Carlo Zaniolo, Stefano Ceri, Christos Faloutsos, Richard T. Snodgrass, V. S. Subrahmanian, Roberto Zicari: Advanced Database Systems. Morgan Kaufmann 1997, ISBN 1-55860-443-X.