Extending SQL for Decision Support Applications

Carlo Zaniolo*

*Course Notes for CS240B
Outline

Part I The problem and the state of the art
Part II Introduction to ATLaS
Part III Decision support applications
Part IV The System and Performance
Part V Conclusions and future directions.
Part I: The Problem

- Databases: where the data is (well most of it)
- But Database Management Systems (DBMSs) do not support well data mining tasks
- Desiderata: Data Mining Query Languages that support ad-hoc mining queries and general data mining
DBMSs and Data Mining

- Many proposals, including:
  - DMQL [Han, Fu, Wang, Koperski, Zaiane: DMDW 1996]
  - Mine operator [Meo, Psaila, Ceri: 1996]
  - M-SQL [Imielinski, Virmani: 1999]

- Difficult technical challenges:
  - No natural way to retrofit SQL with mining operators—as opposed to ROLAP extensions that naturally fit in the (super)group-by syntax
  - Implementation and Performance issues
  - Much diversity in mining tasks: Can one solution fit all?
Data Mining in Object-Oriented DBMSs


- The Question: forget nice SQL extensions, and ask if experts can implement Apriori efficiently in an Object-Relational System such as DB2. An the answer was:
  - Not easily: UDFs are very difficult to write and debug
  - Not as efficient as Cache Mining approaches

- Apriori established as the acid test for the extensibility of DBMSs for data mining tasks.

- Next Question: *is SQL the real cause of these problems and should we instead use other languages for database centric datamining?*
The DATASIFT project uses the logical data language $\mathcal{LDL}++$ to address these problems [Giannotti, Manco, et al. 1999, 2000]

- Both deductive and inductive reasoning needed to support the data mining process
- $\mathcal{LDL}++$ is Turing complete and supports User Defined Aggregates (UDAs)
- Direct C++ implementation of UDAs to solve performance problems

- New Datamining Algebras: The 3W Model [Johnson, Lakshmanan, Ng: VLDB 2000]
Part II: Introducing ATLaS

1. History and main ideas
2. Simple examples: average, minpoints, temporal coalescing after projection
3. Transitive closure computation
A Brief History of ATLaS

1. SQL–AG: extending SQL3 proposal for Aggregates to support ‘early returns’ [1999]
2. LDL++ 5.1: Logic Database Language Monotonic Aggregates: used freely in recursive queries for BoM and greedy algorithms [1999]
3. SADL: Simple Aggregate Definition Language based on SQL. easy to use, but with limited performance and power [2000]
4. AXL: Aggregate eXtension Language: Much more powerful and efficient [2001]
5. ATLaS: table functions and in-memory tables with references [2002]
6. ATLaS: table functions and support for the definition and management of in-memory data structures using SQL [2003]
ATLaS Main Ideas

- Tables as the only data type
- SQL statements as the only statements
- Native Extensibility by letting users introduce new Aggregates and Table functions by coding them in SQL
Defining Aggregates

Aggregates are functions that process a stream of values, on the basis of whether the current item is:

- The first value—INITIALIZE state,
- Every other successive value—ITERATE state,
- The EOF marker—TERMINATE state
- The calling query generates the streams—one for each GROUP BY—and set the states

This way of defining UDAs is similar to that used by Postgres, LDL++, SQL3, etc.

ATLaS aggregates take streams as input but also return streams as output (e.g., online aggregates)
Example: Define 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;
    }
}
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 :
    {
    }
}
SELECT Sex, online_avg(Sal)
FROM employee
WHERE Dept=1024
GROUP BY Sex;
Simple Aggregates in ATLaS

- **SEQ**: an aggregate that appends to each new tuple a consecutive sequence number.
- The **DISTINCT** version of the same (duplicate tuples are ignored).
- To implement that, you must declare a **MEMO** table to memorize old values.
- Then results could be returned:
  - during computation: in the **INITIAL** and **ITERATE** states: nonblocking and monotonic UDA
  - all at the end in the **TERMINATE** state: blocking (and frequently) nonmonotonic UDA
  - Users can exercise high-level control over computation.
The Point and value of Minimum in a sequence of pairs

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;
  }
}
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; }
}
Computation of Transitive Closures

TABLE dgraph(start Char(10), end Char(10)) SOURCE ('mydb');
AGGREGATE reachable(Inode Char(10)) : Char(10)
{
    INITIALIZED: ITERATE: 
    INSERT INTO RETURN VALUES (Inode);
    INSERT INTO RETURN 
    SELECT reachable(end) FROM dgraph 
    WHERE start=Inode;
}

SELECT reachable(dgraph.end) FROM dgraph 
WHERE dgraph.start='000';
Transitive Closures—Cont.

- **reachable** performs a top-down computation (Prolog-like)
- we can also use a memo table to eliminate duplicate results and Prolog’s infinite loops
- We can also express recursion using a bottom-up computation implementing the differential fixpoint algorithm
- In the next slide we show a nonrecursive way, similar to that used by active database triggers.
Incremental Computation of Transitive Closures

In digraph $G$, a node $Y$ is reachable from node $X$ iff there is a simple path from $X$ to $Y$.
Say that $T_C$ is the transitive closure of $G$ to which we now add a new arc $A \rightarrow B$.
Then if for some $X$ and $Y$, $X \rightarrow A \in T_C$ and $B \rightarrow Y \in T_C$, we have four kinds of new simple paths through $A \rightarrow B$ (an arc from the start node to the end node of each path must then be added to $T_C$):

1. $A \rightarrow B$ (Step 1: add $A \rightarrow B$ to $T_C$)
2. $X \rightarrow A \rightarrow B$ (Step 2: add $X \rightarrow B$ to $T_C$)
3. $A \rightarrow B \rightarrow Y$ (Step 3: add $A \rightarrow Y$ to $T_C$)
4. $X \rightarrow A \rightarrow B \rightarrow Y$ (Step 4: add $X \rightarrow Y$ to $T_C$)

But say that we perform these additions serially, and Step 2 produces $T'_C$. Then Steps 3 and 4 can be replaced by:

3’. If $X \rightarrow B \in T'_C$ and $B \rightarrow Y \in T_C$ then add $X \rightarrow Y$ to $T'_C$
Reachable Nodes Incrementally

AGGREGATE \( tclosur(A \text{ Char}(10), B \text{ Char}(10)) \)
\[
: (tcX \text{ Char}(10), tcY \text{ Char}(10))
\]
\[
\{ \quad \text{TABLE tc(snode Char(10), enode: Char(10));}
\]
\[
\text{INITIALIZE: } \quad \text{ITERATE: } \quad \{ 
\]
\[
\quad \text{INSERT INTO tc VALUES(A,B);}
\]
\[
\quad \text{INSERT INTO tc}
\]
\[
\quad \text{SELECT tc.snode, B}
\]
\[
\quad \text{FROM tc WHERE tc.enode=A;}
\]
\[
\quad \text{INSERT INTO tc}
\]
\[
\quad \text{SELECT tc1.snode, tc2.enode}
\]
\[
\quad \text{FROM tc AS tc1, tc2}
\]
\[
\quad \text{WHERE tc1.enode=tc2.snode;}
\]
\[
\}
\]
\[
\text{TERMINATE: } \quad \{ \quad \text{INSERT INTO RETURN SELECT * FROM tc; } \quad \}
\]
\[
\}
\]
\[
The \text{call is: } \quad \text{SELECT tclosur(dgraph.start, dgraph.end)}
\]
\[
\quad \text{FROM dgraph;}
\]
The Power of Streams

- Relationally complete languages cannot express transitive closures
- Recursion had to be added to SQL to express these queries
- Here, we have expressed transitive closure in a non-recursive ATLaS program
- **Conclusion:** a stream-oriented processing model adds significant expressive power to SQL!
- ATLaS taps on this hidden source of power.
- We have in fact proven that ATLAS is Turing Complete.
Nonblocking aggregates are needed for streams
• Every UDA with an empty.terminate clause is nonblocking—also monotonic
• tclosr can be made nonblocking by moving the return to the initialize/iterate states.
• Memory is the second issue for stream-based processing
• Our program only uses one tuple. This is fine if there is no duplicate path (i.e., our graph is a tree) or we do not mind duplicates. Otherwise, we need to store previous pairs in a memo table and add a NOT IN check to the code.