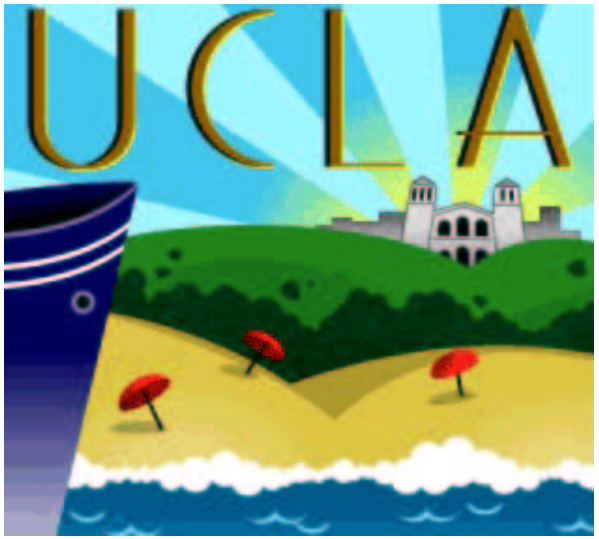
ATLAS: a Small but Complete Extension of SQL for Data Streams and Data Mining

Carlo Zaniolo*



*Joint Work with Haixun Wang, Yan-Nei Law, Richard Luo

Outline

- The problems of SQL with new application areas
- ATLAS: native extensibility and Turing completeness in SQL
- Data Mining Applications
- Data Streams: nonblocking computations
- Data Streams: windows and other new constructs
- Ongoing work and conclusion

Flexibility, Expressivity, Extensibility: not SQL Forte

- New applications have severely challenged SQL, answered with new constructs by standard committees
- O-R extensions help (e.g., user-defined functions written in external languages)
- But critical applications areas cannot be supported by O-R DBs: e.g., Data Mining and Data Streams
- The ATLAS language and system adds native extensibility and Turing completeness to SQL
- Our claim: these features improve DBMSs ability to support new application area. E.G. ATLAS is effective on both Data Mining and Data Streams applications

Extending SQL for DBs: User Define Aggregates (UDAs)

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, SQL3, Aurora, et.— but they use procedural languages.

ATLaS aggregates take streams as input and 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;
   }
}
```

OnLine Average: returning results every 200 tuples

```
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 : {
      }
}
```

Calling UDAs

SELECT **Sex, online_avg(Sal)**
FROM **employee**
WHERE **YoB ¡ 1980**
GROUP BY **Sex;**

The Power of a Stream-oriented Computation on DB Tables

- UDAS make ATLAS Turing Complete—i.e., capable of expressing every possible computable function on the database
- The function is encoded by the TM, the DB becomes the input tape, and then ATLAS is used to implement the instructions of the TM.

Data Streams

Data streams are relations—but fleeting ones:

- Can't wait for the end of input: Blocking queries must be avoided. Results must be returned for each new tuple.
- Short Memory: past grows unbounded and limited memory available (e.g., interval-based or count-based window)

Data Streams and Blocking

We want to characterize (Non)Blocking properties of operators and queries that are fundamental for data streams, by studying:

- **the blocking/nonblocking properties of operators independent of the language in which they are expressed**
- **the blocking/nonblocking properties of UDAs expressed in ATLAS**
- **the class of stream functions expressible by nonblocking operators**
- **Determine the power of Relational Algebra (RA), SQL, and ATLAS for nonblocking computations.**

Blocking Query Operators

- **Blocking Query Operator [PODS02]: *'One unable to produce the first tuple of the output until it has seen the entire input.'*—I.E. until it has detected the end of input.**
  - e.g., ATLAS UDAs which only RETURN **values in** TERMINATE **are blocking**
- **NonBlocking Query Operator: *one that produces all of the tuples in the output before detecting the end of of input***
  - e.g., Atlas UDAs where RETURN **only appears in** ITERATE **and/or** INITIALIZE**.**
- **Partially blocking operators: those that return some of the output before they have detected the end of input, and the rest at that point**
  - e.g., ATLAS UDAs with RETURN **in** TERMINATE **and also in** ITERATE **and/or** INITIALIZE**.**

Computable Functions Expressible using NonBlocking Queries

- Characterize $\mathcal{NB}$: the class of functions expressible via nonblocking computations (independent of any specific language)
- Identify the $\mathcal{NB}$ subsets of various query languages (QLs):
  – what are the $\mathcal{NB}$ operators of RA?
  – what are $\mathcal{NB}$-subsets of SQL and ATLAS?
- Study the expressive power of $\mathcal{NB}$ QLs:
  – what queries can be expressed by the $\mathcal{NB}$ subsets of RA, SQL, or ATLAS?
  – For which query classes are they complete?

The $\mathcal{NB}$-class on Sequences

- Let $L = t_1, \ldots, t_k, \ldots, t_n$ be a sequence of length n. Then $S = t_1, \ldots, t_k$ will be said to be a **presequence** of L, denoted $S \sqsubseteq L$
- $\sqsubseteq$ defines a partial order (reflexive, transitive, and antisymmetric)
- $\mathcal{NB}$ **Characterization:** A function $F(S)$ on a sequence S can be computed using a non-blocking operator, iff F is monotonic with respect to the partial ordering $\sqsubseteq$.

Completeness

- **Proposition:** Every Monotonic function can be expressed by a nonblocking ATLAS UDA (i.e., without TERMINATE).
- So Atlas is also $\mathcal{NB}$-complete.

Presequences and Relations

- Let R_1 and R_2 be sequences. We write $R_1 \subseteq R_2$ to denote that every tuple in R_1 is contained in R_2.
- For presequences: $R_1 \sqsubseteq R_2 \Rightarrow R_1 \subseteq R_2$
- Let $R_1 \equiv R_2$ denote that $R_1 \subseteq R_2 \cap R_2 \subseteq R_1$. Thus R_1 and R_2 are equivalent modulo repetitions (idempotence) and reordering (commutativity). Let $ci(R)$ be the class of sequences equivalent to R. For sets, $\subseteq$ defines a boolean lattice.
- Let R_1 and R_2 be sequences and $ci(R_1)$ and $ci(R_2)$ be their corresponding sets, then $R_1 \sqsubseteq R_2 \Rightarrow ci(R_1) \subseteq ci(R_2)$

Mappings on Relations

Single-Valued functions on relations. Let F be a function on the representations of relations in a certain D. If $F(R_1) \equiv F(R_2)$ for any pair of sequences $R_1 \equiv R_2$ representing the same relation then we say that F is a single-valued function on relations in D.

- Codd's RA operators and expressions of such operators are single valued (SV) functions on their operand relation
- For SQL-2 min and max and the distinct version of sum, count, and avg are SV
- Continuous count, which returns $\{1, \ldots, n\}$ on a set of cardinality n are SV
- Continuous-sum is not SV. In SQL the ALL versions of UNION, EXCEPT, sum, count, avg, and some SQL:1999 OLAP constructs are not

Monotonic Query Operators on Relations

- **Theorem. Single-valued functions on relations can be realized by non-blocking computations iff they are monotonic w.r.t. $\subseteq$.**
- RA: Projection, selection, Cartesian Product, Set Union, and Set Difference.
- In terms of expressive power, RA defines *relational completeness*. The queries expressible in RA are also called FO (first order queries). FO defines the expressive power of several query languages—relational calculus, NR Datalog, etc.
- $\mathcal{NB}$-RA: All above operators but set difference, which is not monotonic
- Only $\mathcal{NB}$-RA can be used on data streams.

Query Languages and Data Streams

- A main focus of research for the last 30 years: how to extend FO languages to support new application domains
- For data streams, only queries in $\mathcal{NB}$-FO can be supported. $\mathcal{NB}$-FO: subsets of monotonic queries in FO.
- Can $\mathcal{NB}$-RA express all the queries in $\mathcal{NB}$-FO?
  – e.g., $R_1 \cap R_2 = R_1 - (R_2 - R_1)$ can also expressed using joins (i.e., Cartesian Product an selection)
- But many monotonic queries expressible in RA cannot be expressed in $\mathcal{NB}$-RA.
- $\mathcal{NB}$-RA is not complete for $\mathcal{NB}$-FO!

Example: Until and Coalesce Queries

- nothing yet

$\mathcal{NB}$-SQL

- Constructs such as EXCEPT, NOT EXIST, and all the aggregates must be left out from $\mathcal{NB}$-SQL
- In addition to queries, such as until and coalesce, we loose queries such:
  In an EMP table find all the departments with more than K employees—HAVING COUM(EMP.*) $>$K.
  This is a monotonic query, which cannot be expressed in $\mathcal{NB}$-SQL
- Recursion (normally not allowed on data stream queries) and a very smart optimizer could solve some of these problems
- Also the problem only affect the nonmonotonic portion of the query. Thus, e.g., for a query $R_1 - R_2$ only R_2 cannot be a stream: there is no problem with R_1.

Query Languages for Data Streams

- Current research projects on data streams are based on SQL and relational languages
- Totally unaware of the loss of expressive power they suffer because of the $\mathcal{NB}$ limitation—and expressive power was already a sore handicap of QLs before this injury
- Punctuation and Windows: interesting and useful concepts that transform some blocking queries into nonblocking ones. However they do not seem to address the problem of expressive power.
- A better solution is needed: actually a quantum leap.

The Quantum Leap

- ATLAS is Turing Complete
- An $\mathcal{NB}$-complete language. One that can express all the monotonic functions (on the database) expressible in by a Turing machine.
- $\mathcal{NB}$-UDAs: those where TERMINATE is missing or empty
- $\mathcal{NB}$-ATLAS: $\mathcal{NB}$-SQL + $\mathcal{NB}$-UDAs
- **Theorem:** $\mathcal{NB}$-ATLAS is NB-Complete.

Data Streams and Windows

Data streams are relations—but fleeting ones:

- Can't wait for the end of input: Blocking queries must be avoided. Results must be returned for each new tuple.
- **Short Memory: past grows unbounded and limited memory available.**
  – Various kinds of synopses and approximations can be used
  – E.g., Interval-based or count-based windows.

Windows like CQL

```
STREAM calls(customer_id Int, type Char(6), minutes Int,
            Tstamp: Timestamp)  SOURCE mystream;
SELECT AVG(S.minutes)
FROM Calls S [  PARTITION BY S.customer id
                RANGE 5 MINUTES PRECEDING
                WHERE S.type = 'Long Distance']
```

- With windows, non-blocking aggregates can be applied to streams
- In ATLAS we have extended the definition of UDAs to work with windows
- We follow SQL:1999 semantics, where windows can only be used as aggregate modifiers.

Joins on Streams

- Call is a stream. *List the length of every call:*

```
SELECT   O1.call_ID,  O2.time - O1.time
FROM     Outgoing O1, Outgoing O2
WHERE    ( AND   O1.call_ID = O2.call_ID
           AND   O1.event = start
           AND   O2.event = end)
```

- For this we would need infinite memory. Some approximation is needed.
- We can use a window of, say, one hour, to hold every calls for one hour. The window is basically treated as a table. The join will return lengths for calls shorter than one hour.
- This is not an efficient approximation. UDAs can be much more efficient