Aggregates in Logic

Carlo Zaniolo,

Computer Science Department

UCLA
DBMSs’ Support for Aggregates

- Only five aggregates in SQL2
- Vendors have added new builtins for data mining—e.g. rollups, datacubes, aggregates for time series
- these are never enough: User-Defined Aggregates (UDAs) are needed
- UDAs are in SQL3, but problems remain and not supported by vendors.
More Flexible UDAs

- Currently aggregates are totally batch-oriented. For instance *Find cities where more than 20 employees live* can easily expressed with **COUNT**.

  But then SQL counts all the 10,000 employees in LA, before checking that this number is larger than 20.

- A small percentage of data is often sufficient for a good estimate of averages. They propose the concept of on-line aggregation to solve this problem [Hellerstein, Hass and Wang]

- To solve this problem we introduce **early returns** in our UDAs.

- The Nonmonotonicity issue.
NonMonotonicity

- **COUNT**: find the suppliers who DO NOT supply red parts ↔ find suppliers where the count of parts they supply is zero.

  MAX and MIN: the highest paid employee is the one for which there is no employee with higher salary.

- Recursive queries are now supported in O/R Databases using techniques and semantics adapted from Deductive Databases: differential fixpoint, magic sets, stratified negation and aggregates.

- SQL queries must be stratified w.r.t. negation and aggregates (SQL3).
A Logical Reconstruction of Aggregates

Inductive Definition of Aggregates

1. **BASE** For a singleton set: \(\text{count}\{x\} = 1; \quad \text{sum}\{x\} = x; \quad \text{max}\{x\} = x\)

2. **INDUCTION** \(\text{sum}(S \sqcup \{x\}) = \text{sum}(S) + x; \quad \text{count}(S \sqcup \{x\}) = \text{sum}(S) + 1\)  
   \(\text{max}(S \sqcup \{x\}) = \text{if } x > \text{max}(S) \text{ then } x \text{ else } \text{max}(S).\)

3. These computations can be easily expressed by logical rules (e.g., 
   the single and multi rules in LDL++)

4. Early returns and Final returns can also be expressed by rules: e.g., 
   \(\text{Avg} = \text{Sum}/\text{Count}\) But we need to enumerate the elements of a set.
Stable Models and Choice Models

- NonMonotonic Reasoning in AI, 30 years of progress: from circumscriptio to stable models.
- But nondeterminism is also an essential facet of stable models. The following programs has dual models.
  \[
  a \leftarrow \neg b. \quad b \leftarrow \neg a.
  \]
- Positive programs with choice can be rewritten as equivalent programs with negated goals displaying a multiplicity of stable models.
  \[
  \text{advisor}(\text{St}, \text{Prof}) \leftarrow \text{student}(\text{St}, \text{Maj}),
  \quad \text{professor}(\text{Prof}, \text{Maj}), \text{choice}(\text{St}, \text{Prof}).
  \]
- **Theorem** [PGZ]: Positive programs with choice define nondeterministic monotonic mappings.
Aggregate Definition in $\texttt{LD\&L++}$

Standard Average:

\[
\begin{align*}
\text{single}(\text{avg}, Y, (Y, 1)). \\
\text{multi}(\text{avg}, Y, (\text{Sum}, \text{Count}), (\text{Sum} + Y, \text{Count} + 1)). \\
\text{freturn}(\text{avg}, Y, (\text{Sum}, \text{Count}), \text{Avg}) \quad \leftarrow \quad \text{Avg} = \text{Sum}/\text{Count}.
\end{align*}
\]

On line average: returns a value every 100 samples.

\[
\begin{align*}
\text{ereturn}(\text{avg}, X, (\text{Sum}, \text{Count}), \text{Avg}) \quad \leftarrow \quad \text{Count mod 100 = 0}, \text{Avg} = \text{Sum}/\text{Count}.
\end{align*}
\]

Using the aggregate remains the same:

\[
\begin{align*}
p(\text{DeptNo}, \text{avg}(\text{Sal})) \quad \leftarrow \quad \text{empl}(\text{Ename}, \text{Sal}, \text{DeptNo}).
\end{align*}
\]
Aggregates Formal Definition

\[ p(\text{avg}(Y)) \leftarrow d(Y). \]

We replace this by

\[ p(Y) \leftarrow \text{results}(\text{avg}, Y). \]

where \text{results}(\text{avg}, Y) is derived from \( d(Y) \) by

- the chain rules,
- the \text{cagr} rules and
- the \text{return} rules.
The chain rules are those with choice that place the elements of \( d(Y) \) into an order-inducing chain.

Then, the \texttt{cagr} rules perform the inductive computation by calling the \texttt{single} and \texttt{multi} rules as follows:

\[
\begin{align*}
\text{cagr}(\text{AgName}, Y, \text{New}) & \leftarrow \text{chain}(\text{nil}, Y), Y \neq \text{nil}, \text{single}(\text{avg}, Y, \text{New}). \\
\text{cagr}(\text{AgName}, Y2, \text{New}) & \leftarrow \text{chain}(Y1, Y2), \text{cagr}(\text{AgName}, Y1, \text{Old}), \\
& \quad \text{multi}(\text{AgName}, Y2, \text{Old}, \text{New}).
\end{align*}
\]

Thus, the \texttt{cagr} rules are used to memorize the previous results, and to apply (i) \texttt{single} to the first element of \( d(Y) \) (i.e., for the pattern \text{chain}(\text{nil}, Y)) and (ii) \texttt{multi} to the successive elements. The return
rules are as follows:

\[
\text{results}(\text{AgName}, \text{Yield}) \leftarrow \text{chain}(Y_1, Y_2), \text{cagr}(\text{AgName}, Y_1, \text{Old}), \\
\text{ereturn}(\text{AgName}, Y_2, \text{Old}, \text{Yield}).
\]

\[
\text{results}(\text{AgName}, \text{Yield}) \leftarrow \text{chain}(X, Y), \neg\text{chain}(Y, \_), \\
\text{cagr}(\text{AgName}, Y, \text{Old}), \\
\text{freturn}(\text{AgName}, Y, \text{Old}, \text{Yield}).
\]

Therefore we first compute \textbf{chain}, and then \textbf{cagr} that applies the \textbf{single} and \textbf{multi} to every element in the chain.

Concurrently, the first \textbf{results} rule produces all the results that can be generated by the application of \textbf{ereturn} to each element of the chain.
Monotonic Aggregates

The final returns are computed by the second \texttt{results} rule that calls \texttt{freturn} once the last element in the chain (i.e., the element without successors) is detected.

This is the only rule using negation; in the absence of \texttt{freturn} this rule can be removed yielding a positive choice program that is monotonic!

Thus every aggregate with only early returns is \textit{monotonic with respect to set-containment} and can be used freely in recursive rules. \textit{Monotonic aggregates}

To define a new aggregate, the user must write the \texttt{single}, \texttt{multi}, \texttt{ereturn} and \texttt{freturn} rules; the remaining rules are built in the system.
Monotonic Aggregates: no \texttt{freturn}

Continuous count:

\begin{verbatim}
single(mcount, \textit{Y}, 1).
ereturn(mcount, \textit{Y}, \textit{Old}, \textit{New}) \leftarrow \textit{Old} = \text{nil}, \textit{New} = 1
ereturn(mcount, \textit{Y}, \textit{Old}, \textit{New}) \leftarrow \textit{Old} \neq \text{nil}, \textit{New} = \textit{Old} + 1.
\end{verbatim}

You can define \texttt{msum} in a similar fashion.
mcount and msum

- Say that instead of count and sum we use mcount, msum, which returns a new partial count or sum for each new element in the set.
- Thus for a set of cardinality 5 mcount returns: 1, 2, 3, 4, 5.
- If we add a new element to the set mcount returns: 1, 2, 3, 4, 5, 6.
- mcount is monotonic and deterministic. But msum is a nondeterministic (i.e., multivalued) monotonic mapping.
- New aggregates are conducive to more efficient algorithms.
Monotonic Aggregates—Applications

The query, ‘Find all departments with more than 7 employees’ can be expressed as follows:

\[
\text{count\_emp}(D\#, \text{mcount}(E\#)) \leftarrow \text{emp}(E\#, \text{Sal}, D\#).
\]
\[
\text{large\_dept}(D\#) \leftarrow \text{count\_emp}(D\#, \text{Count}), \text{Count} = 7.
\]

Find all departments with less than 7 employees:

\[
\text{small\_dept}(D\#, \text{Dname}) \leftarrow \text{dept}(D\#, \text{Dname}), \neg \text{large\_dept}(D\#).
\]
Monotonic Aggregates—Applications

Join the party: Some people will come to the party no matter what, and their names are stored in a \texttt{sure(Person)} relation. But others will join only after they know that at least $K = 3$ of their friends will be there. Here, \texttt{friend(P,F)} denotes that F is P’s friend.

\begin{align*}
\text{willcome}(P) &\leftarrow \text{sure}(P), \\
\text{willcome}(P) &\leftarrow \text{c_friends}(P,K), K \geq 3, \\
\text{c_friends}(P,mcount(F)) &\leftarrow \text{willcome}(F), \text{friend}(P,F).
\end{align*}
Basic semi-naive computation yields:

```prolog
sure(mark).
sure(tom).
sure(jane).
friend(jerry, mark).
friend(penny, mark).
friend(jerry, jane).
friend(penny, jane).
friend(jerry, penny).
friend(penny, tom).

willcome(mark).
willcome(tom).
willcome(jane).
c_friends(jerry, 1).
c_friends(penny, 1).
c_friends(jerry, 2).
c_friends(penny, 2).
c_friends(penny, 3).
willcome(penny).
c_friends(jerry, 3).
willcome(jerry).
```