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 Recostruction of Aggregates

Inductive Definition of Aggregates

1. *BASE* For a singleton set: $count(\{x\}) = 1; \quad sum(\{x\}) = x;$ $max(\{x\}) = x$
2. *INDUCTION* $sum(S \sqcup \{x\}) = sum(S) + x; \quad count(S \sqcup \{x\}) = sum(S) + 1$
   $max(S \sqcup \{x\}) = if x > max(S)$ then x else $max(S)$.
3. These computations can be easily expressed by logical rules (e.g., the *single* and *multi* rules in $\mathcal{LDL}$++)
4. Early returns and Final returns can also be expressed by rules: e.g., Avg= Sum/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 circumscription to stable models.
- But nondetermism is also an essential facet of stable models. The following programs has dual models.

  $\texttt{a} \leftarrow \neg\texttt{b}.\ \texttt{b} \leftarrow \neg\texttt{a}.$
- Positive programs with choice can be rewritten as equivalent programs with negated goals displaying a multiplicity of stable models.

$$\begin{aligned}\texttt{advisor}(\texttt{St},\texttt{Prof}) \leftarrow\ & \texttt{student}(\texttt{St},\texttt{Maj}),\\ & \texttt{professor}(\texttt{Prof},\texttt{Maj}),\texttt{choice}((\texttt{St}),(\texttt{Prof})).\end{aligned}$$

- **Theorem** [PGZ]: Positive programs with choice define nondeterministic monotonic mappings.

Aggregate Definition in $\mathcal{LDL}$++

Standard Average:

$\texttt{single}(\texttt{avg}, \texttt{Y}, (\texttt{Y}, \texttt{1})).$
$\texttt{multi}(\texttt{avg}, \texttt{Y}, (\texttt{Sum}, \texttt{Count}), (\texttt{Sum} + \texttt{Y}, \texttt{Count} + \texttt{1})).$
$\texttt{freturn}(\texttt{avg}, \texttt{Y}, (\texttt{Sum}, \texttt{Count}), \texttt{Avg}) \leftarrow \texttt{Avg} = \texttt{Sum}/\texttt{Count}.$

On line average: returns a value every 100 samples.

$\texttt{ereturn}(\texttt{avg}, \texttt{X}, (\texttt{Sum}, \texttt{Count}), \texttt{Avg}) \leftarrow \texttt{Count mod 100} = \texttt{0}, \texttt{Avg} = \texttt{Sum}/\texttt{Count}.$

Using the aggregate remains the same:

$\texttt{p}(\texttt{DeptNo}, \texttt{avg}\langle\texttt{Sal}\rangle) \leftarrow \texttt{empl}(\texttt{Ename}, \texttt{Sal}, \texttt{DeptNo}).$

Aggregates Formal Definition

$$\texttt{p}(\texttt{avg}\langle\texttt{Y}\rangle) \leftarrow \texttt{d}(\texttt{Y}).$$

We replace this by

$$\texttt{p}(\texttt{Y}) \leftarrow \texttt{results}(\texttt{avg}, \texttt{Y}).$$

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

- the chain rules,
- the `cagr` rules and
- the`return` rules.

Aggregates: Definition

The chain rules are those with choice that place the elements of $\mathtt{d}(\mathtt{Y})$ into an order-inducing chain.

Then, the **cagr** rules perform the inductive computation by calling the **single** and **multi** rules as follows:

$$
\begin{aligned}
\mathtt{cagr}(\mathtt{AgName}, \mathtt{Y}, \mathtt{New}) \leftarrow\ & \mathtt{chain}(\mathtt{nil}, \mathtt{Y}), \mathtt{Y} \neq \mathtt{nil}, \mathtt{single}(\mathtt{avg}, \mathtt{Y}, \mathtt{New}).\\
\mathtt{cagr}(\mathtt{AgName}, \mathtt{Y2}, \mathtt{New}) \leftarrow\ & \mathtt{chain}(\mathtt{Y1}, \mathtt{Y2}), \mathtt{cagr}(\mathtt{AgName}, \mathtt{Y1}, \mathtt{Old}),\\
& \mathtt{multi}(\mathtt{AgName}, \mathtt{Y2}, \mathtt{Old}, \mathtt{New}).
\end{aligned}
$$

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

$$
\begin{aligned}
\texttt{results}(\texttt{AgName}, \texttt{Yield}) \leftarrow\ & \texttt{chain}(\texttt{Y1}, \texttt{Y2}), \texttt{cagr}(\texttt{AgName}, \texttt{Y1}, \texttt{Old}),\\
& \texttt{ereturn}(\texttt{AgName}, \texttt{Y2}, \texttt{Old}, \texttt{Yield}).\\
\texttt{results}(\texttt{AgName}, \texttt{Yield}) \leftarrow\ & \texttt{chain}(\texttt{X}, \texttt{Y}), \neg\texttt{chain}(\texttt{Y}, \_),\\
& \texttt{cagr}(\texttt{AgName}, \texttt{Y}, \texttt{Old}),\\
& \texttt{freturn}(\texttt{AgName}, \texttt{Y}, \texttt{Old}, \texttt{Yield}).
\end{aligned}
$$

Therefore we first compute `chain`, and then `cagr` that applies the `single` and `multi` to every element in the chain.

Concurrently, the first `results` rule produces all the results that can be generated by the application of `ereturn` to each element of the chain.

Monotonic Aggregates

The final returns are computed by the second `results` rule that cals `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 `freturn` this rule can be removed yielding a positive choice program that is monotonic!

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

To define a new aggregate, the user must write the `single`, `multi`, `ereturn` and `freturn` rules; the remaining rules are built in the system.

Monotonic Aggregates: no `freturn`

Continuous count:

$$\texttt{single}(\texttt{mcount}, \texttt{Y}, 1).$$
$$\texttt{multi}(\texttt{mcount}, \texttt{Y}, \texttt{Old}, \texttt{New}) \leftarrow \quad \texttt{New} = \texttt{Old} + 1.$$
$$\texttt{ereturn}(\texttt{mcount}, \texttt{Y}, \texttt{Old}, \texttt{New}) \leftarrow \texttt{Old} = \texttt{nil}, \texttt{New} = 1$$
$$\texttt{ereturn}(\texttt{mcount}, \texttt{Y}, \texttt{Old}, \texttt{New}) \leftarrow \texttt{Old} \neq \texttt{nil}, \texttt{New} = \texttt{Old} + 1.$$

You can define 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:

$$\texttt{count\_emp}(\texttt{D\#}, \texttt{mcount}\langle\texttt{E\#}\rangle) \leftarrow \texttt{emp}(\texttt{E\#}, \texttt{Sal}, \texttt{D\#}).$$
$$\texttt{large\_dept}(\texttt{D\#}) \leftarrow \texttt{count\_emp}(\texttt{D\#}, \texttt{Count}), \texttt{Count} = \texttt{7}.$$

Find all departments with less than 7 employees:

$$\texttt{small\_dept}(\texttt{D\#}, \texttt{Dname}) \leftarrow \texttt{dept}(\texttt{D\#}, \texttt{Dname}), \neg\texttt{large\_dept}(\texttt{D\#}).$$

Monotonic Aggregates—Applications

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

$$
\begin{aligned}
\texttt{willcome}(\texttt{P}) &\leftarrow \texttt{sure}(\texttt{P}). \\
\texttt{willcome}(\texttt{P}) &\leftarrow \texttt{c\_friends}(\texttt{P}, \texttt{K}), \texttt{K} >= \texttt{3}. \\
\texttt{c\_friends}(\texttt{P}, \texttt{mcount}\langle \texttt{F} \rangle) &\leftarrow \texttt{willcome}(\texttt{F}), \texttt{friend}(\texttt{P}, \texttt{F}).
\end{aligned}
$$

```
sure(mark).
sure(tom).
sure(jane).

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

Basic semi-naive computation yields:

```
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).
```