University of California
Los Angeles

User-Defined Aggregates in Database Languages

A thesis submitted in partial satisfaction
of the requirements for the degree
Master of Science in Computer Science

by

Punit Bhargava

1998

The thesis of Punit Bhargava is approved.

D. Stott Parker

Mario Gerla

Carlo Zaniolo, Committee Chair

University of California, Los Angeles

1998

To my parents . . .

who were always there for me

Table of Contents

List of Figures

List of Tables

Acknowledgments

I would like to thank my advisor Dr. Carlo Zaniolo for his constant guidance, encouragement and support.

Abstract of the Thesis

User-Defined Aggregates in Database Languages

by

Punit Bhargava

Master of Science in Computer Science

University of California, Los Angeles, 1998

Professor Carlo Zaniolo, Chair

Applications, such as OLAP, data warehousing, and data mining, are steering today's database industry. These applications process highly complex queries and make extensive use of complex aggregates such as online aggregation and data cubing. Current SQL2 compliant database systems cannot support these complex aggregates. In fact, SQL2 only supports five aggregates and provides no mechanism for defining new aggregates. Constructs for specifying new aggregates (i.e., user defined aggregates) are part of the forthcoming SQL3 standards. In this thesis, we propose SQL-AG, a language and a system to support user-defined aggregate in SQL. The proposal extends SQL3 user-defined aggregates and removes some of their limitations to allow the definition of on-line aggregates, data cubes and other aggregates used in data mining. This report describes the design and implementation of the SQL-AG system. We begin with an introduction to the problem, followed by an analysis of the design issues related to SQL-AG. All details of the design and implementation of SQL-AG, along with a discussion of the implementation environment based on Oracle Corporation's server and PL/SQL, are presented.

CHAPTER 1

Introduction

With the increasing demands of new database applications such as OLAP, data warehousing and data mining, a need arises for an extension of current database systems, which implement SQL2 [DD93] standards. These applications in fact use new and complex aggregates extensively. SQL2 supports only five built-in aggregates and provides no mechanism to define new aggregates. Approaches based on ad-hoc extensions and procedurally defined aggregates have been tried, but have proven unsatisfactory due to their rigidity and complexity. A better approach is to have *extensible, user-definable aggregates.* Such an approach will let the user define complex aggregates, as needed to support specific classes of new applications.

1.1 A New Wave of Database Applications

With increases in the volume of data maintained by the industry there has been an emergence of a new set of applications. These applications can be broadly described as data warehousing, OLAPs and data mining applications, though there is no fine line separating the three.

1.1.1 Data Warehousing and OLAP

Data warehousing [Wid95] are means to facilitate fast, reliable and less expensive way of collecting and analyzing important information. A data warehouse provides integrated access to the information collected from multiple, distributed and heterogeneous databases. Over the past few years there has been a great interest in this topic in the database industry. Most leading vendors provide some tool for data warehousing. Though there has been high demand for data warehousing tools, most available tools are inflexible and have limited features. These tools are mostly used for decision support queries (often with aggregates), On-line analytical processing (OLAP), and data mining.

OLAP is a technology that enables users to gain insight into data through fast, consistent and interactive access to a wide variety of possible views of information. These applications involve dynamic multidimensional analysis of data. The key requirement of these applications is the ability to aggregate across many sets of dimensions. Such a computation is a performance bottleneck for these applications. Aggregate extensions of particular interest in this context are data cubes and online aggregation.

1.1.1.1 Data Cube

In Data cube [GBA96, RS97, AAD96, ZDN97, HRU96] , aggregates compute along multiple dimension and granularities. These are very useful in OLAP or multidimensional data analysis applications. For example an application could perform an aggregate computation along the time dimension, say every year, every season and so on. A straightforward way to compute such aggregates is by grouping for every subset, but is not viable due to bad performance. User-defined aggregates allow the user to define such aggregates in a much simpler and efficient manner.

1.1.1.2 Online aggregation

Unlike traditional approach of entering the aggregate computation in batch mode and waiting till it is completely processed, online aggregation is the aggregates that let the user to obtain partial results and allows the user to stop the computation in between. These type of aggregates are useful in applications where an exact value is not needed and an approximate will suffice. The requirements of online aggregation and the details of an extension of POSTGRES95 to support online aggregation are discussed in [HHW97].

1.1.2 Data Mining

Data mining is a technology to extract information of predictive use hidden in the database. Along with other uses, data mining is useful for *automated predictions of trends and behaviors* and *automatic discovery of previously unknown patterns.* A few of the commonly used data mining algorithm are:

- Decision trees,
- Genetic algorithm,
- Artificial neural networks, and
- Rule induction.

Details of the above uses and algorithm are out of the scope of this report.

Data Mining uses complex aggregates. The following are a few applications using aggregates:

- Mining for associations,
- Data Classification, and
- Time series applications.

1.1.2.1 Mining for Associations

Data mining evolved as a set of application relating to discovery of relevant hidden information in databases, especially discovery of association rules [AS94, MPC96]. As an application is basket data analysis, where the data of purchased products (referred as items), is collected and associations between items are computed. An association rule is of the form $X \Rightarrow Y$ where X and Y are two sets of items. The frequency of such an observation in data gives the *strength* of the rule. As an example for a probable rule:

$$\mathsf{citizen(x,c),official\_language(c,l)} \rightarrow \mathsf{speaks(x,l)}$$

the relevance could be measured by:

$$C_f = \frac{|(\pi_{1,2}C \bowtie O) \wedge S|}{|(\pi_{1,2}C \bowtie O)|}$$

The closer C_f is to 1 the stronger the confidence in the rule. This could be computed in LDL++ using *avg* rather than *count*.

```
overlap(avg<W>) ← citizen(X,Ct),official_language(Ct,L)
                      if(speaks(X,L) then W = 1 else W = 0).
```

The above LDL++ code computes a function overlap. This function computes the average value of W. W is assigned 0 or 1 for every pair of tuples: (X,Ct) from table citizen and (Ct,L) from table official_language. It assigns 1 to W if speaks(X,L) is true for that pair of tuple else assigns 0. Now C_f is the ratio of the count of pair of such tuples for which speaks is true to the count of all the pairs. Which could be expressed in terms of W as simply avg<W>. Hence the above code computes C_f using average. Since averages can be estimated, this can be computed using online aggregation.

1.1.2.2 Data Classification

Data Classification [Wei98a, Wei98b] finds use in data analysis and pattern recognition. Data classification is a form of supervised learning. Given attributes and training instances, it outputs predictions. The attributes consist of independents, dependents and targets. An example set of attributes are:

[Day, Outlook, Temperature, Humidity, Wind, **PlayTennis**]

Here the target is PlayTennis and it is dependent on the attributes like Outlook, Temperature, Humidity and Wind. An example of a training instance is:

(D1, Sunny, Hot, High, Weak, **Yes**)

Data classification outputs the expressions of predication made from dependents to targets. One of the expressions could be:

(Outlook=Sunny) $\Rightarrow$ (PlayTennis =Yes)

It has been shown that probability could be used for prediction. *Naive Bayesian* [Elk97] classifier which uses probability for prediction has shown significant results in terms of predictive performance. The next paragraph describes the fundamentals of Naive Bayesian classifier.

Let A_1, A_2, ..., A_n be attributes, with discrete values, used to predict a discrete class C. Given an example with observed attribute values a_1 through a_k, the optimal prediction is class value c such that $Pr(C = c|A_1 = a_1 \wedge \ldots \wedge A_k = a_k)$ is maximal. This probability, using Bayes' rule can be rewritten as:

$$\frac{Pr(A_1 = a_1 \wedge \ldots \wedge A_k = a_k) * P(C = c)}{P(A_1 = a_1 \wedge \ldots \wedge A_k = a_k)}$$

Here P(C=c) can be easily computed based on the training data and $P(A_1 = a_1 \wedge \ldots \wedge A_k = a_k)$ is independent of c. Using assumptions on the independence of the attributes $Pr(A_1 = a_1 \wedge \ldots \wedge A_k = a_k)$, can be written as:

$$Pr(A_1 = a_1|C = c)Pr(A_2 = a_2|C = c) \ldots Pr(A_k = a_k|C = c)$$

where each probability can be estimated from the training data using:

$$\hat{Pr}(A_1 = a_1|C = c) = \frac{count(A_j = a_j \wedge C = c)}{count(C = c)}$$

Here if the training data is in table T with $A_1, \ldots, A_k$, C as fields, then we can compute $count(C = c)$ as

```
select C, Aj, count(*)
from T
group by C, Aj;
```

thus requiring separate computation for each column. Let Table 1.1 be used as the training data. Now we want to predict which of the following will be more probable:

P(Play=Yes|[Sunny, Hot, High, Weak]) = ?

or

P(Play=No|[Sunny, Hot, High, Weak]) = ?

To compute the above probabilities using naive bayesian approach we will have to compute the following probabilities:

Day	Outlook	Temp	Humidity	Wind	PlayTennis
D1	Sunny	Hot	High	Weak	No
D2	Sunny	Hot	High	Strong	No
D3	Overcast	Hot	High	Weak	Yes
D4	Rain	Mild	High	Weak	Yes
D5	Rain	Cool	Normal	Weak	Yes
D6	Rain	Cool	Normal	Strong	Yes
D7	Overcast	Cool	Normal	Strong	No
D8	Sunny	Mild	High	Weak	No
D9	Sunny	Cool	Normal	Weak	Yes
D10	Rain	Mild	Normal	Weak	Yes
D11	Sunny	Mild	Normal	Strong	Yes
D12	Overcast	Mild	High	Strong	Yes
D13	Overcast	Hot	Normal	Weak	Yes
D14	Rain	Mild	High	Strong	No

Table 1.1: Tennis Table

P(Play=Yes), P(Play=No)

P(Outlook=Sunny|Play=Yes), P(Outlook=Sunny|Play=No)

P(Humidity=High|Play=Yes), P(Humidity=High|Play=No)

P(Temp=Hot|Play=Yes), P(Temp=Hot|Play=No) and

P(Wind=Weak|Play=Yes), P(Wind=Weak|Play=No)

Or in other words to compute the following counts:

count(Play=Yes), count(Play=No)

$\text{count(Outlook=Sunny} \wedge \text{Play=Yes)}$, $\text{count(Outlook=Sunny} \wedge \text{Play=No)}$

$\text{count(Humidity=High} \wedge \text{Play=Yes)}$, $\text{count(Humidity=High} \wedge \text{Play=No)}$

$\text{count(Temp=Hot} \wedge \text{Play=Yes)}$, $\text{count(Temp=Hot} \wedge \text{Play=No)}$ and

$\text{count(Wind=Weak} \wedge \text{Play=Yes)}$, $\text{count(Wind=Weak} \wedge \text{Play=No)}$

These *count s* can be computed by using the SQL code given earlier in this section. It can be observed that to compute the above predictions it requires 10 passes of the table *Tennis*. But these can be computed using user-defined aggregates in a single pass by computing all the counts together.

1.1.2.3 Time Series Analysis

Time series applications have data as a sequence of events, with each event having a time of occurance. These type of applications finds prominent use in financial analysis and analysis by retailers. Financial analysts frequently try to predict interest rate fluctuations or stock performance based on a series of preceding events. Time series analysis use temporal aggregation [ZM97], details of which are given in next paragraph.

A few kind of temporal aggregation used for time-series analysis are:

- Running aggregates : Aggregate computed for each point in the sequence, based on all the earlier events, e.g. price of a mutual fund from its inception.
- Moving-window aggregates: Aggregate computed based on the last n events in the sequence, e.g. average price of a stock since quarterly dividend.
- Temporal grouping: The time line is partitioned into consecutive time intervals and aggregate computed for each interval, based on events in the

interval, e.g. price of a stock average over month, quarter, year.

As we will see user-defined aggregates can be used to express temporal aggregates, under the assumption that the data is sorted according to time.

1.2 Commercial DBMS and SQL3

In response to their market, some commercial DBMS have implemented additional aggregates in their systems. For instance **RISQL** [Red96] is Red Brick's extension of SQL supporting complex queries and aggregates. RISQL supports pre-defined aggregates such as *MovingAvg*, *MovingSum* etc. It also allows the user to define queries with parameters using *Create Macro*.

The commercial world is expected to move towards user-defined aggregates, once the SQL3 standards, which are still in development, will be introduced. SQL3 [Iso96a, Iso96b, Mat96] proposes a concept of SQL-invoked routines termed as *iterative routines* to allow user-defined aggregates.

An *iterative routine* is composed of three routines :

- initialize routine, the routine that gives an initial value to the aggregate.
- iterate routine, the one that computes the intermediate value each time a new record of the table is accessed.
- terminate routine, the routine that returns the final value computed for the aggregate.

The three routines are user-defined by means of some procedural language.

For example a user can define an iterative routine for average as :

AGGREGATE FUNCTION average(**IN NUMBER**)

RETURNS NUMBER

STATE NUMBER

INITIALIZE average-init

ITERATE average-iterate

TERMINATE average-terminate

Here average-init, average-iterate and average-terminate are also user-defined.

1.2.1 Limitations of SQL3

Though SQL3 proposes this approach, it doesn't specify how the approach should be implemented. Also, the users can not define any online aggregate within this framework, where the terminate routine is expected to return the results.

1.3 Previous Research

1.3.1 POSTGRES

Work on aggregates was done at the EECS department of UC Berkley, where the database management system, POSTGRES [RS87, SR86, SRH90], was designed and implemented. POSTGRES allows the user to specify aggregates in terms of state transition functions, where an aggregate is defined as a state, and state transition functions are used to define the computation of aggregates.

There are two types of state functions: **sfunc1** and **sfunc2**. Here **sfunc1** are state transition functions that compute the new state depending on the old state and value of the record. **Sfunc2** are state transition functions that compute the new state dependent only on the old state. Sum is an example of an aggregate that uses only **sfunc1**, and count is an aggregate that uses only **sfunc2**. Average requires both a function to compute the running sum and a function to compute the running count. When all the records are processed, the final answer for the aggregate is sum divided by the count.

User uses *initcond* to define the initial state of the aggregate. *Finalfunc* is used to define the computation once all the records are processed.

Following example aggregate *Average* is written in POSTGRES and is taken from the POSTGRES user manual (http://pluto.xtech.ru/postgres95/users-guide/pg95user.html)

```
create aggregate my_average (
        sfunc1 = int4pl, – sum
        basetype = int4,
        stype1 = int4,
        sfunc2 = int4inc, – count
        stype2 = int4,
        finalfunc = int4div, – division
        initcond1 = '0',
        initcond2 = '0')

Select my_average(salary) as emp_average from EMP;
```

In the above example, int4pl, int4inc and int4div are built into POSTGRES. Int4pl adds the new value of the record to the current state and int4inc adds one to the state variable each time a record is processed. Int4div computes the average by dividing the first state variable be second state variable.

1.3.2 LDL++ Experience

The logical data language LDL is a rule based language for databases [NT89, Zan88] developed at Microelectronics and Computer Technology Corporation. LDL++ supports lists, sets, complex objects, recursion and extensible aggregates. In LDL++ every aggregate function f on a set is inductively defined by:

1. BASE: $f(\{\mathrm{x}\})$ on a singleton set $\{\mathrm{x}\}$
2. INDUCTION: for any other set $\mathrm{S}' = \mathrm{S} \sqcup \{\mathrm{x}\}$ derive the value of $f(\mathrm{S}')$ from those of S and $f(\{\mathrm{x}\})$

We can write count, sum as:

count: $\mathsf{f}(\{\mathsf{x}\}) = 1$ and $\mathsf{f}(\mathsf{S}\sqcup \{\mathsf{x}\}) = \mathsf{f}(\mathsf{S}) + \mathsf{f}(\{\mathsf{x}\})$
sum: $\mathsf{f}(\{\mathsf{x}\}) = \mathsf{x}$ and $\mathsf{f}(\mathsf{S}\sqcup \{\mathsf{x}\}) = \mathsf{f}(\mathsf{S}) + \mathsf{f}(\{\mathsf{x}\})$

In this extension of LDL++ the first function is called *single*, and the second one is called *multi*. We can rewrite count and sum as

```
count:
        single(X,1).
        multi(S,Old,New) ← New = Old + 1.
sum:
        single(X,X).
        multi(S,Old,New) ← New = Old + S.
```

1.3.2.1 UCLA Extension

A major extension of LDL++, user-defined aggregates [ZAO93] has been recently implemented at UCLA. Rather than waiting for the result till the end of the set, partial results could be shown to the user using *early returns*. The following example returns average every 10^{th} element.

```
avg:
        single(X,(X,1)).
        multi((X,1),(Sum,Count),Sum+X, Count+1)).
        return((X,1),(Sum,Count), Avg) ← mod(Count,10) = 0, Avg = Sum/Count.
```

Early returns allow the support of online aggregation in LDL++.

This implementation of user-defined aggregates has shown successful results in terms of ease of use and strong expressive power. A few examples that have been successfully implemented over this extension are:

- Online average,
- Online median estimation,
- Data Cube aggregates and
- Temporal aggregation.

One can refer to http://www.cs.ucla.edu/˜czdemo/javademo for details.

1.4 SQL-AG Approach

SQL-AG allows the user to define new aggregates in SQL. It is an extension of SQL supporting *extensible, user-definable aggregates*. The system was developed using Oracle DBMS and PL/SQL as the implementation environment. All features of SQL-AG are described in Chapter 3.

In this approach the user first defines the aggregate using a specification language SQL-AG, described in Section 3.3. Then using SQL-AG the user generates an aggregate in SQL and can use the aggregate in an equivalent way as a built-in aggregate. This approach is similar to what SQL3 has proposed, but several extensions have been introduce to make it more powerful using the lessons learnt from LDL++. SQL-AG approach also uses an *iterative routine*. The *iterative routine* is composed of three routines namely **single**, **multi** and **produce**. As described in more detail in later chapters, these three routines are similar to **initialize**, **iterate** and **terminate** routines of SQL3. The following are the key differences in the two approaches:

- unlike **initialize**, **single** computes an initial value based on the first record,
- unlike **terminate**, **produce** returns partial results depending on intermediate states, rather than just a final result depending on the final state.

Apart from other benefits, these extensions allow SQL-AG to support online-aggregates, which can not be supported by SQL3 approach.

1.5 Outline of the Thesis

The report is organized as follows. The next chapter analyses the design requirements on SQL-AG. The chapter discusses the syntactic requirements and the requirements posed by online aggregation. Chapter 3 describes the SQL-AG system, the specification language and the pre-compiler. The discussion is followed by the details of the usage of the SQL-AG system. Chapter 4 gives a few example aggregates written in SQL-AG. Chapter 5 talks about the implementation environment followed by some implementation details in Chapter 6. We conclude with Chapter 7. Specification of an example in SQL-AG is given in appendix A. Appendix B contains a few sample calls to a user-defined aggregate in SQL-AG.

CHAPTER 2

Design Requirements

This chapter describes the design requirements for SQL-AG. First, the common usage of the aggregates are listed, followed by the requirements posed by online aggregation. The last section gives the SQL-AG solution with design details relating to its requirements.

2.1 Syntactic Requirements

Current commercial DBMS SQL2 have five built-in aggregates and do not provide any mechanism to define new aggregates. Here is an example usage of the predefined aggregate *avg*.

```
SELECT avg(salary)
FROM emp;
```

The above SQL query computes the average salary of the employees from the *emp* table. This computes a single, final average. In case of a very large table, the user might be interested in partial results, rather than just the final result. Say that the user wants to find the average salary every 5^{th} record. Then he/she will define a new aggregate, say *average*, that computes average every 5^{th} record. And will like to use this newly defined aggregate as follows:

```
SELECT average(salary)
FROM emp;
```

The user might also use *average* in a group-by query, such as

```
SELECT deptno, average(salary)
FROM emp
GROUP BY deptno;
```

or with a where clause condition.

```
SELECT average(salary)
FROM emp
WHERE salary > 1000;
```

The above queries will compute the aggregate in the same way as *avg* but will return partial results, in this case every 5^{th} record. SQL-AG implementation of *average* is given in section 4.2.

The user can also use the pre-defined aggregates in nested queries. For example the following query finds the name of the employee with salaries greater than the average salary of their respective department.

```
SELECT e1.ename
FROM emp e1
WHERE e1.sal ≥
   (SELECT avg(e2.sal)
   FROM emp e2
```

```
    WHERE e2.deptno = e1.deptno);
```

These queries require the aggregates to return partial results. Hence the user-defined aggregates should also be able to return partial results. SQL-AG implementation of the above sub-query is given in section 4.5.

The above usages can be written in two generic forms as:

```
SELECT [GroupFields,] aggregate(ResponsibleFields)
FROM InputTable
[WHERE WhereClause1]
[GROUP BY GroupFields]
;
or
SELECT Fields
FROM InputTable1
WHERE Fieldi [> | < | =]
    (SELECT aggregate(ResponsibleFields)
    FROM InputTable2
    [WHERE WhereClause2])
;
```

Where InputTable is the table on which the aggregate is being computed, ResponsibleFields are the fields of InputTable on which aggregate is computed and GroupFields are the fields of InputTable on which user wants to group by. Records of InputTable satisfying the $\mathsf{WhereClause}_1$ condition are used in the computation of the aggregate. $\mathsf{InputTable}_1$ and $\mathsf{InputTable}_2$ are name of the input tables. Fields and Field_i are fields of $\mathsf{InputTable}_1$. $\mathsf{WhereClause}_2$ is a condition dependent on zero or more fields of the input tables. One can easily note that ResponsibleFields, GroupFields and Fields are set of fields, rather than just a field and GroupFields and Fields can be an empty set also. $\mathsf{WhereClause}_1$ can also be null.

The syntactic requirements posed by the above usages can be summarized as the requirements that SQL-AG aggregates must :

- take table name, where clause condition, set of responsible fields and group fields as input,
- allow null group fields,
- allow null where clause condition and
- allow output either on screen or in a user-defined table.

To support return of partial results the user-defined aggregates should be interactive and persistent. The aggregates created should store information persistently and return partial-results as the computation progresses.

2.2 Comparison with SQL3 Requirements

SQL-AG approach is similar to SQL3 but it has been extended to support other applications such as online aggregation, data cube and data mining applications. Like SQL3 the aggregate will be defined in terms of *iterative routines*. Hence the user will define the three iterative routines for the computation of the aggregate. This requires SQL-AG to accept single (similar to initialize), multi (similar to iterate) and produce (similar to terminate) from the user. It is also required that the intermediate value of an aggregate be stored internally. This intermediate value might be a tuple of values, rather than a single value. For example in case of the aggregate *count* only the count of records need to be stored internally whereas, in case of *average*, count of records along with current sum need to be stored. So it is required that SQL-AG should allow user to specify this tuple.

2.3 Language Requirements

The requirements posed by the usage of the aggregates and by the approach chosen can be recast in terms of the requirements on the language used for the implementation as follows. The implementation must:

- Supports easy and simple constructs to access database,
- Support for defining the three functions single, multi and produce,
- Allows suitable mechanism for defining the tuple for storing the intermediate values of an aggregate

Keeping the above requirements in mind, PL/SQL was chosen as the implementation environment. Apart from other advantages given in section 5.9, it fulfills the above requirements. PL/SQL is Oracle Corporation's procedural extension to SQL. It provides a tight integration to SQL and hence provides a simple and easy access to database server. It supports functions and procedure definition, hence allows defining single, multi and produce. Along with simple types PL/SQL also supports record and table types, that can be used to store intermediate states of an aggregate.

CHAPTER 3

SQL-AG System

This chapter deals with the design details of the SQL-AG system. The first section talks about the basic idea behind the SQL-AG system. Subsequent sections deal with the details of the architecture of the system, the language and pre-compiler. We then give some examples to clarify the use of SQL-AG.

3.1 Basic Idea

In their simplest manifestation, aggregates can be described as the end result of a calculation over every record of a table. In a more advanced form, aggregates return partial results during the computation. In other words an aggregate is calculated each time a record of the table is processed, and it reaches a final value once all the records have been processed.

We associate a state with an aggregate. Initially the aggregate has some state $S_{initial}$ depending on the first record of the table. Then with each record processed the state of the aggregate changes depending on the value of the fetched record. Finally after all the records of the table are processed, the aggregate has state S_{final}. The result is then produced based on all the states the aggregate takes. In simple aggregates like sum, max, count etc. the result is produced based on only S_{final}. Whereas in online aggregates results are produced every i^{th} record and hence depend on all the intermediate states S_j of the aggregate.

The process of the calculation of an aggregate can be written in pseudo-code as:

```
Fetch first record, Calculate S_initial.
for every new record fetched
    Compute new S_i based on the record fetched and S_i
    Output partial result depending on S_i
Output final result based on S_final.
```

We now bring out the notion of single, multi and produce. The calculation of the aggregates can be summarized by just these three functions. Single is the action to be performed to calculate $S_{initial}$ from the first record. Multi is the action to be performed to compute state new S_i, given the next record and state S_i. Produce is the action to be performed to produce the result depending on any intermediate state S_i or state S_{final}. We can rewrite the computation of the aggregate in terms of single, multi and produce as:

```
Fetch first record in r, S_i = single(r), produce(S_i)
for every new record r fetched
    S_i = multi(r, S_i)
    produce(S_i)
produce(S_final)
```

A generic tool to generate code for an aggregate thus will accept the definition of single, multi and produce and use it to generate the code for that aggregate.

3.2 Architecture

The main component of SQL-AG is its pre-compiler that is used to generate the code for an aggregate, given its specification in SQL-AG.

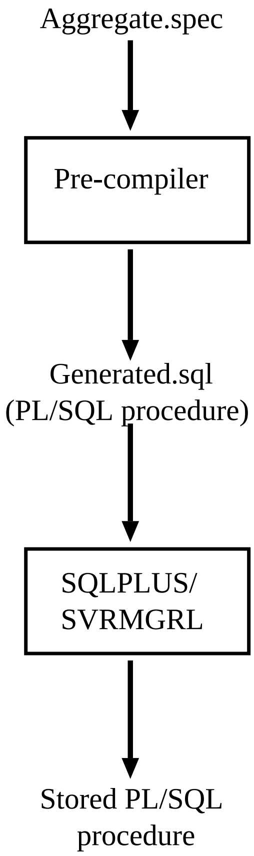

Figure 3.1: Creating stored PL/SQL procedure

Figure 3.1 gives a general idea of how the tool works. The user writes the aggregate using SQL-AG, the specification language. Using the pre-compiler, a PL/SQL procedure is generated in the file generated.sql. Then either using *sqlplus* [1] or *svrmgrl* [2] a PL/SQL stored procedure is created. This procedure could then be used in any further PL/SQL program.

[1] *The client application used to access Oracle server*
[2] *The Server Manager application used to manage Oracle server*

3.3 Defining a New Aggregate

The language accepted by the pre-compiler to define the aggregate is also termed as SQL-AG. This section describes this language.

The following is a simple aggregate *count* defined in SQL-AG. This aggregate finds the number of records in a table.

Example 1 *Specification of aggregate* **Count** *in SQL-AG*

```
aggregate count;
    Outputdefn output(v_result number);
    type internalrecord is record (counter number);
    procedure single (v_f1 number, n OUT internalrecord) is
    begin
        n.counter:=1;
    end ;
    procedure multi (v_f1 number, o internalrecord,
        n IN OUT internalrecord) is
    begin
        n.counter:= o.counter + 1;
    end ;
    procedure produce (o internalrecord, flag varchar2) is
    begin
        if ( flag = 'normal_flag') then
            null;
        elsif (flag = 'eof_flag') then
                Output(o.counter);
        end if;
```

end ;

.

Here **Count** is the name of the aggregate in the above example. User defines the prototype of Output function, which is used in procedures Single, Multi and Produce to output any information either on the screen or a user-defined table. In the above example since the user only wants to output a single field, the count of the records, Output is defined with a single argument v_result. InternalRecord is a record type that is used to store the state of the aggregate. It consists of all the variables used to keep the information about the state of the aggregate. In case of **count** user only wants to store the current count of records, hence defines internalrecord with only one field, counter.

Single, Multi and Produce defined here are similar to the single, multi and return function introduced in first section. Given the first tuple, Single computes the initial state of the aggregate. In **count**, tuple is a singleton and Single returns the initial state in its second argument, which is of type internalrecord. Given the old state and a new tuple, Multi computes the new state. In **count** the oldstate, new tuple and new state are in o, v_f1 and n respectively. Produce computes the result depending on the current state. Produce contains a set of actions to be performed depending on the value of the flag. It should contain actions for at least two values of the flag, 'normal_flag' and 'eof_flag' that , respectively, specify the action to be performed after each record (but not the last record) and at the end of the table. Other flags can be used by the user for defining other actions to be performed in Single and Multi, this might be useful for debugging purposes. For **count** where the user only wants to output the total number of records in a table , no value is to be produced at each record and only the counter value has to be returned at the end of the table. Hence output is used only for the flag value 'eof_flag'.

The types supported by SQL-AG are varchar2, chars, rawdata, number, date and bool. Of these types varchar2, chars and rawdata require sizes being variable size types.

The user could also use three pre-defined types t_stringtable, t_numbertable and t_datetable for any variables declared in SQL-AG. These types are pre-defined as the tables of varchar2 (string), number and date correspondingly and are useful for aggregates that require the state to be a table. Examples such as *maxname* described in subsection 4.3 use one of these types.
The following is the main structure of a SQL-AG program used to define a new aggregate:

Aggregate aggregate_name ;

OutputDefn Output [(parameter[, parameter, ...])];

Type InternalRecord is record (field_name datatype, [field_name datatype, ...]);

[variable_name datatype; ...]

Procedure Single ([parameter , ...], variable_name **InternalRecord**);

[local declarations]

Begin

executable statements

END;

Procedure Multi ([parameter , ...], variable_name **InternalRecord**,

variable_name **InternalRecord**);

[local declarations]

Begin

executable statements

END;

```
Procedure Produce (variable_name InternalRecord, flag varchar);
    [local declarations]
Begin
    executable statements
END;
```

.

where *parameter* stands for the following syntax:

```
parameter_name datatype
```

and *local declarations* stands for:

```
[ variable_name datatype; ...]
```

In the above syntax, Single, Multi and Produce are PL/SQL procedures. Also, the term "executable statements" denotes any PL/SQL code and "datatype" denotes any PL/SQL type. Detail description of the syntax of PL/SQL is given in Chapter 5.

3.4 Compiling a New Aggregate

Once the user has defined the aggregate using SQL-AG, he/she can call the precompiler to generate the following stored PL/SQL procedure :

```
aggregate_name ( InputTableName, ResponsibleFields, WhereClause, GroupFields,
```

OutputTable, 'OrderBy'default is non-Order By);

Therefore the user calls the pre-compiler *Aggregate* on the file filename.spec. The pre-compiler uses a file text.sql that contains aggregate independent code to generate the procedure in a file generated.sql. The user gives the following command :

Aggregate < filename.spec

Here filename.spec contains the specification of the aggregate.
The Pre-compiler accepts a program written in SQL-AG and collects the following information:

- name of the aggregate,
- argument list for the procedure Output,
- type declaration of internalrecord type,
- definitions of the procedures Single, Multi and Produce,
- code of other subprograms defined by user.

Using this information collected the pre-compiler generates the PL/SQL procedure.

3.5 User-Defined Aggregates in SQL-AG Queries

As described in chapter 2 the aggregate can be used as:

SELECT aggregate(ResponsibleFields)

```
    FROM InputTableName
    [WHERE condition]
or
    SELECT [GroupFields,] aggregate(ResponsibleFields)
    FROM InputTableName
    [WHERE condition]
    GROUP BY GroupFields
;
```

The first one above is an aggregate calculation without any group-by fields, and second one is aggregate calculation with group-by.

The user can specify the first one in the following way for the created stored PL/SQL procedure:

```
aggregate_name ( InputTableName, ResponsibleFields, WhereClause, 'NULL', OutputTable);
```

Here depending on the OutputTable, if 'STDOUT' then the output is dumped on the screen, else the OutputTable is created and the output is inserted into that table. Also the user can shorten the above procedure in case the OutputTable is 'STDOUT' and write as:

```
aggregate_name ( InputTableName, ResponsibleFields, WhereClause);
```

(This is because GroupFields and OutputTable has default values as 'NULL' and 'STDOUT').

WhereClause contains the condition for the where clause, for which it accepts any expression or 'EMPTY. If this is 'EMPTY', the above command can be further reduced to:

```
aggregate_name ( InputTableName, ResponsibleFields);
```

For the second template above there are two ways the user can compute the aggregate. The user either can pre-sort the table (by using the **order-by** statement of SQL for fetching the records of the InputTable) and compute the aggregate. Or the table is fetched in the existing order, storing for each group-by value internally. User can specify either ways in the following manner:

```
aggregate_name ( InputTableName, ResponsibleFields, WhereClause, GroupFields,
  OutputTable, 'OrderBy');
aggregate_name ( InputTableName, ResponsibleFields, WhereClause, GroupFields,
  OutputTable);
```

(Note: Non **order-by** is the default way.)

The above ways are the defined ways the user is suppose to use the aggregate. All the calls are legal calls in any PL/SQL code. For just making calls to aggregates :

```
BEGIN
  aggregate_name ( InputTableName, ResponsibleFields, WhereClause, GroupFields,
     OutputTable);
END
```

would suffice.

3.6 Online Version

This section gives the detail of O-SQL-AG the *online version* of SQL-AG. This version allows the user to define the online version of an aggregate. The only difference between SQL-AG and O-SQL-AG is the pre-compiler. The code generated by the two versions is different. O-SQL-AG supports all the functionality of SQL-AG, with slight variations in the usage. O-SQL-AG uses a pre-defined package Aggpackage. Packages are a pre-defined object type in PL/SQL allowing the user to support global variables and grouping of sub programs. The file package.sql contain the PL/SQL source code for the package aggpackage.

3.6.1 Generating Aggregate

Similar to SQL-AG user uses the pre-compiler *Aggregate* and the text file text.sql to generate the procedure in a file generated.sql. User gives the following command :

```
Aggregate < filename.spec
```

Here filename.spec contains the specification of the aggregate in SQL-AG. Along with a package aggregatename_package, following stored PL/SQL procedure is created:

```
aggregate_name ( InputTableName, ResponsibleFields, WhereClause, GroupFields,
  OutputTable, 'OrderBy'default is non-Order By);
```

3.6.2 Using the Created Procedure

In O-SQL-AG to support online aggregation, packages are used to store information persistently. Aggpackage contains the aggregate independent information and Aggregatename_package contains the aggregate dependent information. To use the created stored procedure in any PL/SQL code, user need to initialize aggpackage first using:

```
aggpackage.init;
```

and need to call the following procedure once the aggregate is used.

```
aggpackage.close;
```

Following is a simple call to the created aggregate.

```
BEGIN
  aggpackage.init;
  aggregate_name ( InputTableName, ResponsibleFields, WhereClause, GroupFields,
      OutputTable);
  aggpackage.close;
END
```

The default usage of the aggregate is non-online. To use the aggregate as online, user has to set aggpackage.online to true. The variable aggpackage.eof_table is true if the whole table has been processed and the variable aggpackage.output_result is true if the function has returned a result. Variables aggregatename_package.gvariable_name contain the value of the partial result. (Here variable_name is the name of the variable used in defining the prototype of Output function). The variable aggpackage.supress is used to supress all the output on screen and aggpackage.debug is used to output extra debug information. Next chapter contains an example computing online average and an example using online aggregation in evaluating a sub-query.

CHAPTER 4

Applications of SQL-AG

In this chapter we look into a few examples. First some sample tables are given. For each example a brief description along with the specification program for the aggregate is given. Then examples of PL/SQL code using the aggregates along with their output are given.

4.1 Tables

The tables used in the given examples are defined as:

```
emp = (
        empno number,
        ename varchar2(10),
        sal number,
        deptno number);

sales = (
        year number,
        season varchar2(10),
        month varchar2(5),
        day number,
```

```
sale number,
deptid number);
```

Data for these tables are given in table 4.1 and table 4.2.

empno	ename	sal	deptno
7369	SMITH	5000	20
7499	ALLEN	2850	30
7521	WARD	1250	30
7566	JONES	2975	20
7654	MARTIN	1250	30
7698	BLAKE	2850	30
7782	CLARK	2450	10
7788	SCOTT	3000	20
7839	KING	5000	10
7844	TURNER	1500	30
7876	ADAMS	1100	20
7900	JAMES	950	30
7902	FORD	3000	20
7934	MILLER	1300	10

Table 4.1: Emp Table

4.2 Average

This example finds the average for any field of a table and outputs the current estimate every 5^{th} record. The following is the specification of the aggregate in SQL-AG.

year	season	month	day	sale	deptid
1992	winter	feb	6	409	8
1992	winter	feb	26	239	5
1992	spring	mar	7	318	5
1992	summer	jun	23	394	6
1992	summer	jul	1	167	4
1992	summer	aug	22	176	1
1992	summer	aug	24	256	5
1992	fall	oct	13	481	1
1992	fall	oct	21	116	8
1992	fall	nov	11	136	2

Table 4.2: Sales Table

Example 2 *Specification of aggregate* **Average** *in SQL-AG*

```
aggregate average;
    Outputdefn output(v_result number);
    type internalrecord is record (
        total number,
        counter number);
    procedure single (v_f1 number, newvar OUT internalrecord) is
    begin
        newvar.total:=v_f1;
        newvar.counter:=1;
    end ;
    procedure multi (v_f1 number, oldvar internalrecord,
        newvar IN OUT internalrecord) is
```

```
begin
    newvar.total:= oldvar.total + v_f1;
    newvar.counter:= oldvar.counter + 1;
end ;
procedure produce (oldvar internalrecord, flag varchar2) is
begin
    if ( flag = 'normal_flag') then
        if ((oldvar.counter mod 5)=0) then
            Output(oldvar.total/oldvar.counter);
        end if;
    elsif (flag = 'eof_flag') then
        if ((oldvar.counter mod 5)!=0) then
            Output(oldvar.total/oldvar.counter);
        end if;
    end if;
end ;
```

.

By invoking the pre-compiler a stored PL/SQL procedure named average is created. Following is a call to the procedure along with the output. (These commands were given at the *sqlplus* command line prompt)

```
SQL> begin average('emp','sal');end;
  2    /
select sal from emp
2665
```

```
2812.5
2462.5

PL/SQL procedure successfully completed.
```

More example calls to the procedure are given in appendix B.

4.3 Name of Employees with Maximum Salary

This example finds the name of all the employees with maximum salary. Since in this case the state of the aggregate needs to be a record with one of the elements being the table, a table of type t_stringtable (table_of_name) is declared to store the names of the employees with current maximum. The specification in SQL-AG for the aggregate is given in appendix A. After using SQL-AG a stored PL/SQL procedure named maxname is created. Example calls to the procedure along with the output follows.

```
SQL> begin maxname('emp','sal,ename');end;
  2   /
select sal,ename from emp
5000 SMITH
5000 KING

PL/SQL procedure successfully completed.

SQL> begin maxname('emp','sal,ename','deptno','STDOUT','OrderBy');end;
```

```
    2     /
Create table InternalTable (DEPTNO number, Id number)
select deptno from InternalTable where Id = :g_id
select deptno,sal,ename from emp
10 5000 KING
20 5000 SMITH
30 2850 ALLEN
30 2850 BLAKE

PL/SQL procedure successfully completed.
```

4.4 Rollups for Data Cubes

This example is one of the set of applications SQL-AG is desired for. As mentioned in Chapter 1, data cubes are applications in which the aggregates are computed along several dimensions and multiple granularities. In this example the data used was sorted along the time dimension and the aggregate computes the total sales by Day, Month, Season and Year. The specification in SQL-AG for the aggregate is given in appendix A. After using SQL-AG a stored PL/SQL procedure named datacube is created. An example call to the procedure along with the output follows.

```
SQL> begin datacube('sales','year,season,month,day,sale');end;
    2     /
select year,season,month,day,sale from sales
1992 winter feb 6 ? ? ? 409
```

1992 winter feb 26 ? 648 648 239

1992 spring mar 7 ? 318 318 318

1992 summer jun 23 ? ? 394 394

1992 summer jul 1 ? ? 167 167

1992 summer aug 22 ? ? ? 176

1992 summer aug 24 ? 993 432 256

1992 fall oct 13 ? ? ? 481

1992 fall oct 21 ? ? 597 116

1992 fall nov 11 2692 733 136 136

PL/SQL procedure successfully completed.

4.5 Sub Query

This example exemplifies the use of O-SQL-AG aggregates in defining sub-queries. This query finds the name of the employee with salary greater than the average salary of his/her department. The following is the complex SQL query for this example.

```
SELECT e1.ename
FROM emp e1
WHERE e1.sal ≥
   (SELECT avg(e2.sal)
   FROM emp e2
   WHERE e2.deptno = e1.deptno)
```

Now we redefine average2 aggregate in SQL-AG, and then use PL/SQL to use the defined aggregate in sub-query. The following is the specification of the aggregate in SQL-AG.

Example 3 *Specification of aggregate* **Average2** *in SQL-AG*

```
aggregate average2;
    Outputdefn output(v_result number);
    type internalrecord is record (
        total number,
        counter number);
    procedure single (v_f1 number, newvar OUT internalrecord) is
    begin
        newvar.total:=v_f1;
        newvar.counter:=1;
    end ;
    procedure multi (v_f1 number, oldvar internalrecord,
        newvar IN OUT internalrecord) is
    begin
        newvar.total:= oldvar.total + v_f1;
        newvar.counter:= oldvar.counter + 1;
    end ;
    procedure produce (oldvar internalrecord, flag varchar2) is
    begin
        if ( flag = 'normal_flag') then
            null; elsif (flag = 'eof_flag') then
            Output(oldvar.total/oldvar.counter);
        end if;
    end ;
```

.

The following PL/SQL code in file sub-query.sql uses average2 to solve the problem.

```
DECLARE
   cursor c is select ename, deptno from emp;
BEGIN
   FOR e1 in c
   LOOP
      aggpackage.init;
      aggpackage.online:=TRUE;
      aggpackage.supress:=TRUE;
      WHILE NOT(aggpackage.eof_table)
      LOOP
         average2('emp','sal','deptno='||e1.deptno);
      END LOOP;
      IF e1.sal ≥ average2_package.gv_result THEN
         dbms_output.put_line(e1.deptno||' '||e1.ename);
      END IF;
      aggpackage.close;
   END LOOP:
END;
```

In the above code *cursor c* is used to retrieve records from the table *emp*. For each record in *emp* average salary for the department is computed. The partial results are in average2_package.gv_result.

Following is a call to the above PL/SQL code along with the output. (These commands were given at the *sqlplus* command line prompt)

```
SQL> @sub-query
20 SMITH
30 ALLEN
30 BLAKE
10 KING

PL/SQL procedure successfully completed.
```

4.6 Approximate Average

This example finds approximate average using aggregate average that returns average every 5^{th} record. Average is defined using O-SQL-AG using the spec given in subsection 4.2. The approximate average is returned when the change in average for the last 10 records is 0.03% or less.

The following PL/SQL procedure computes the approximate average, using the average aggregate.

```
CREATE or REPLACE PROCEDURE approx_avg(p_TABLE VARCHAR2, field VARCHAR) IS
  no_of_rec number:=0;
  old_avg number;
  new_avg number;
  counter NUMBER :=1;
BEGIN
    aggpackage.init;
    aggpackage.online:=TRUE;
    aggpackage.supress:=TRUE;
```

```
            average(p_table,field);
            old_avg:= average_package.gv_result;
            WHILE (NOT(aggpackage.eof_table) AND (no_of_rec<10)
            LOOP
            counter:= counter +1;
            average(p_table,field);
            new_avg:=average_package.gv_result;
            if (((new_avg/old_avg) > 0.997)AND((new_avg/old_avg)<1.003)) then
                no_of_rec := no_of_rec + 1;
            else
                no_of_rec := 0;
            END IF;
            old_avg:=new_avg;
            END LOOP;
            dbms_output.put_line('Average: '||new_avg);
            dbms_output.put_line('Records: '||counter *5);
            aggpackage.close;
        END LOOP:
    END;
```

Following is a call to approx_avg. (These commands were given at the *sqlplus* command line prompt)

```
SQL> begin approx_avg('employee','salary');end;
 2 /
Average: 50220.3469
Records: 735
```

```
PL/SQL procedure successfully completed.
```

The table used contained 100,000 randomly generated 100,000 records. The actual average for the given data was 50045.8429. The approximate average was computed within an error margin of 0.34% after 735 out the 100,000 records were visited.

CHAPTER 5

Implementation Environment

This chapter deals with the implementation environment used for SQL-AG.

5.1 Oracle

Oracle [AC97] is the most commonly used database server in industry. It provides a comprehensive environment to support database applications. Along with good performance, Oracle also provides a tightly integrated procedural extension of SQL, PL/SQL. In view to these features, Oracle was chosen as the database for SQL-AG, and PL/SQL was chosen as the implementation environment for SQL-AG. The following sections give a descriptive overview of PL/SQL.

5.2 PL/SQL

PL/SQL [Urm97] is Oracle Corporation's procedural extension to SQL. PL/SQL offers the functionality of a procedural language like C and simple constructs to access data from Oracle database. Since PL/SQL is integrated with the database server, its code can be processed quickly and efficiently.

5.3 Main Features

PL/SQL is a block-structured language. Like Pascal it treats a procedure, function or a program as a logical block that might contain any number of nested sub-blocks. A logical block is made up of declarations and statements, and the declarations are local to the block.

```
[DECLARE
   – declarations]
BEGIN
   – statements
[EXCEPTION
   – handlers]
END;
```

As shown above, a PL/SQL block is made of three parts: a declarative part, an executable part, and an exception handling part. Any error or a warning condition is called an exception in PL/SQL. Only the executable part is necessary.

5.4 Data Types

Following is a list of a few of the simple data types handled by PL/SQL.

- fixed size types: number, boolean, date, rowid, mlslabel, and
- variable size types: chars, varchar2, rawdata.

Apart from the simple data types, PL/SQL also allows record and table types. One can define a record or a table type in the following way.

TYPE type_name **IS RECORD** (

field_name$_1$ field_type$_1$, ..., field_name$_n$ field_type$_n$);

TYPE *type_name* **IS TABLE OF** element_type

[NOT NULL] [INDEX BY BINARY_INTEGER];

Then the type_name defined can be used to declare variables of these record or table types.**NOT NULL** is used to specify if the table accepts only non null values. **INDEX BY BINARY_INTEGER** is used to declare tables that can be accessed by an integer index, as in one dimensional arrays of Pascal.

5.5 Control Structures

This section discusses the constructs to define the flow of control in a PL/SQL program. PL/SQL provides conditional control, iterative control and sequential control constructs.

5.5.1 Conditional Control

PL/SQL has the IF statement for executing a statement conditionally. There are three forms of IF statements : IF-THEN, IF-THEN-ELSE and IF-THEN-ELSIF.

The simplest form of IF statement is shown below. It executes a sequence of statements only if the *condition* is true.

IF *condition* **THEN**

sequence_of_statements;

END IF;

The second from of IF statement executes first sequence of statements if the *condition* is true otherwise executes the alternate sequence of statements.

```
IF condition THEN
    sequence_of_statements1;
ELSE
    sequence_of_statements2;
END IF;
```

As an alternate to *case* statement of Pascal, PL/SQL has IF-THEN-ELSIF statement. One of the mutually exclusive alternatives are executed depending on which of the given conditions is true.

```
IF condition1 THEN
    sequence_of_statements1;
ELSIF condition2 THEN
    sequence_of_statements2;
.
.
.
ELSIF conditionn THEN
    sequence_of_statementsn;
ELSE
    sequence_of_statementsn+1;
END IF;
```

5.5.2 Iterative Control

PL/SQL provides a LOOP construct for execution of a sequence of statements multiple number of times. There are three forms of LOOP statements: LOOP, FOR-LOOP and WHILE-LOOP.

The basic form of LOOP statement allows unconditional iteration over a number of statements.

```
LOOP
    sequence_of_statements;
END LOOP;
```

To complete the LOOP statement unconditionally or conditionally, the programmer can use EXIT, or EXIT-WHEN statements.

```
LOOP
    ...
    [EXIT [WHEN condition];]
    ...
END LOOP;
```

The WHILE-LOOP iterates over the sequence of statements, each time evaluating the condition, executing the statements in case the condition is true and completing execution when the condition is false. FOR-LOOP iterates over the statements for a specified range of integers (Between lower_bound and higher_bound). Codes for both are given below:

```
WHILE condition LOOP
    sequence_of_statements;
END LOOP;

FOR variable IN [REVERSE] lower_bound..higher_bound LOOP
    sequence_of_statements;
```

```
END LOOP;
```

5.5.3 Sequential Control

PL/SQL provides GOTO and NULL statements for the sequence control. These are not that used in common PL/SQL programming. GOTO allows user to change the flow of control unconditionally and NULL statement explicitly specifies inaction. Code for both are given below:

```
BEGIN
   ...
   GOTO insert_row;
   ...
   <<insert_row>>
   ...
END;

BEGIN
   ...
   NULL;
   ...
END;
```

5.6 Accessing Oracle Database

PL/SQL supports all SQL data manipulation commands (except EXPLAIN PLAN), transaction control commands and functions. PL/SQL supports INSERT, UP-

DATE, DELETE, SELECT and LOCK TABLE commands for manipulating Oracle data. For controlling the transactions it supports COMMIT, ROLLBACK, SAVEPOINT, and SET TRANSACTION commands. All the group functions or in other words the commonly supported aggregate functions in SQL are also supported by PL/SQL (avg, count, max, min, stddev, sum, and variance are supported by Oracle SQL). Several books on PL/SQL provide a detailed treatment of its syntax and usage, for instance [Urm97].

5.7 Functions and Procedures

Subprograms in PL/SQL are similar to any other procedural language. The syntax of defining function and procedure is given below.

FUNCTION name [(*parameter*[, *parameter*, ...])] **RETURN** datatype **IS**

  [local declarations]

BEGIN

  executable statements

[**EXCEPTION**

  exception handlers]

END;

PROCEDURE name [(*parameter*[, *parameter*, ...])] **IS**

  [local declarations]

BEGIN

  executable statements

[**EXCEPTION**

  exception handlers]

END;

where *parameter* stands for the following syntax:

parameter_name [**IN** | **OUT** | **IN OUT**] datatype

where datatype is any type, either simple or user-defined. IN, OUT, IN OUT define the behavior of formal parameters. IN parameter lets you pass values to subprogram, but a value cannot be assigned to IN parameters. OUT parameter lets the subprogram to return values to the caller. IN OUT variables allows the user to pass initial values to the subprogram and return updated values to the caller.

5.8 Dynamic SQL

For executing dynamically defined queries, PL/SQL has defined dbms_sql Package. The following steps are to be taken for executing a dynamic query.

- Open a cursor,
- Parse the string containing the query,
- Bind variables,
- Execute the query,
- Fetch record from the result of the query.

5.9 Advantages of PL/SQL

PL/SQL offers the following advantages over any other procedural language:

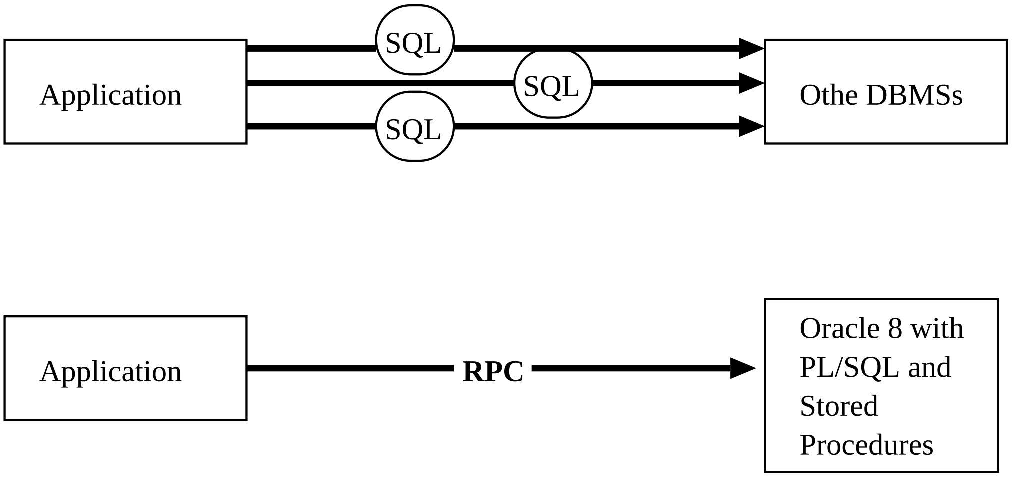

Figure 5.1: PL/SQL Performance

- support for SQL,
- closer integration with Oracle leading to better performance, and
- support for object-oriented programming.

Among the other advantages, performance of PL/SQL is mainly due to its architecture. Without PL/SQL, Oracle must process SQL statements one at a time. Each SQL statement results in a new call to Oracle and leads to higher performance overhead. In case of a network environment, this means a transmit and receive for each SQL statement, increasing the overhead significantly. However, with PL/SQL, an entire block of statements can be sent to Oracle one at a time. A PL/SQL engine resides either at the server-end or at the client-end in an application development tool. In the former case processing of an entire block of PL/SQL statements takes place at the server end and hence drastically reduces the communication between the application and Oracle server. Figure 5.1 shows implementations of an application in Oracle without PL/SQL and with PL/SQL using remote procedure call (RPC).

The main disadvantages of PL/SQL is its lack of portability. This was not viewed as a major drawback in this proof-of-concept experiment. As discussed in the conclusion other languages e.g. C++ and JDBC should be considered in the future.

CHAPTER 6

Implementation Details

This chapter deals with the details of the implementation of SQL-AG system. The first section gives the algorithm of the generated PL/SQL procedure. The second section talks about the main data structures used, and the final section gives some performance comparison.

6.1 Algorithm

This section gives the algorithm of the generated PL/SQL procedure in pseudo-code. The code generated can be divided into aggregate dependent and aggregate independent code. Most of the generated code is independent of the aggregate and is kept in the file 'text.sql'. Apart from some variable declaration, definition of output function and calls to the single, multi and produce procedures, the code is aggregate independent.

```
Aggregate_name (InputTable, ResponsibleField, WhereClause, GroupField,
    TempTable, OrderBy)

    Initialize variables
    if TempTable != 'STDOUT' then
        Create TempTable
```

```
end if
if Non-null GroupField then
    Create Internal Table
end if
Fetch GroupField, ResponsibleField from InputTable satisfying WhereClause
    into genericvariable and elementlist
if Non-null GroupField then
    if OrderBy used then
        loop until all records are fetched
        if first record then
            call single(elementlist, state)
            call produce(state, 'normal_flag');
        else
            if Change in groupfield value then
              call produce(state, 'eof_flag');
              call single(elementlist, state);
              call produce(state, 'normal_flag');
            else
              call multi(elementlist, state, newstate);
              state:=newstate
              call produce(state, 'normal_flag');
            end if
        end if
        end loop
        call produce(state, 'eof_flag');
    else
```

```
        loop until all records are fetched
        Find the hash value for the given set of group field values
        if new entry then
            call single(elementlist, state)
            Insert the groupfield value and hash value into SQLInternalTable
            Insert state with hash value as index into PL/SQLInternalTable
        else
            call multi(elementlist, PL/SQLInternalTable(hashvalue), state);
        end if
        call produce(state, 'normal_flag');
        end loop
        For all groupfield values
            call produce(SQLInternalTable(hashvalue),'eof_flag');
    end if
else
    loop until all records are fetched
    if first record then
        call single(elementlist, state);
    else
        call multi(elementlist, state, newstate);
        oldinternalvar:=newstate;
    end if
    call produce(state, 'normal_flag');
    end loop
    call produce(state, 'eof_flag');
end if
```

6.2 Data Structures Used

6.2.1 Generic Record

Being a strongly structured language PL/SQL disallow any pointer and void types. This makes it more suitable for use as a procedural extension of SQL in databases, providing a more safe environment. But this also poses a strict requirement on the data structures used in any PL/SQL code. Only way to allocate dynamic memory in PL/SQL is tables.

SQL-AG allows dynamic definition of the group fields. This requires that variables be dynamically allocated for fetching the group field values from the table. A simple, but not straight forward, approach is to define a generic record type. A record type that contains fields of every type and a field type-field of type number. Only one of the fields of the record is used to store value at a time depending on the value of the type-field field. In simple words we have a record type that imitates *union* of C. A table of generic record type allows to have any number and any type of group fields.

6.2.2 Internal Table

The usage of aggregates with the *group-by* clause poses a requirement of storing the state of the aggregate internally. In case the aggregate is being computed with Non-null groupfield then, an internal table has to be maintained to store the state of the aggregate associated with each group-by value. PL/SQL doesn't allow tables with elements whose types will be defined dynamically. Since the user can group-by multiple number of fields and the type of the group-by is defined dynamically, internal table cannot be implemented just by PL/SQL tables. SQL tables can be created dynamically hence could be used for internal tables. But access to SQL tables is very slow as compared to PL/SQL tables.

So a hybrid solution with a better performance was implemented. Both SQL and PL/SQL tables are used to store the internal table. For any group field value a unique hash value is computed. The SQL table stores the group field value and hash value generated for that group field value. The PL/SQL table keeps the state of the aggregate associated with a group field indexed by its hash value.

To make this more clear, consider the following example. Let there be n group fields, S and P be the SQL and PL/SQL table respectively. Let $g_1, g_2, \ldots, g_n$ be a particular groupfield value that occurs i times in the table. Then S will contain one entry ($g_1, g_2, \ldots, g_n$, hash($g_1, g_2, \ldots, g_n$)) for this group field value. And at any time P (hash($g_1, g_2, \ldots, g_n$)) will have the state associated with $g_1, g_2, \ldots, g_n$.

6.3 Performance

6.3.1 Performance Measurement in Oracle

Oracle 8.0 provides a tool *tkprof* to analyze the performance of any application written for Oracle. For using *tkprof* utility a trace of the session containing required information has to be generated. To take a trace of any session, the following command has to be given at the command line prompt of *sqlplus* or *svrmgrl*:

```
SQL> alter session set sql_trace = true;
```

This command will create a trace file on the completion of the session. The following command is given on the unix shell to extract the timings from the trace file using the *tkprof* utility.

```
tkprof trace-file trace.out
```

Just typing tkprof on the unix shell will give help on tkprof, giving the various options that could be used. The trace.out file contains the cpu time and elapsed time spent for each sql command. The timings are broken down into parsing, execution and fetching time.

6.3.2 Analysis

In this section, we give the performance of some example aggregates written in SQL-AG. The examples used for the performance analysis are *average* and *rollups* or *datacube*. *Average* is similar to the builtin function *avg*. Since PL/SQL has overhead for accessing tables, a comparison of these examples with direct SQL is unreasonable. A comparison of these examples with simple traversal of table using PL/SQL is shown in Table 6.1. Also the time of the aggregate average written directly in PL/SQL without using dynamic SQL and SQL-AG is given as *avg_opt*.

In the above experiments the number of records used were 100,000.

Aggregate	Time Spent (in sec)
traversal	12
avg_opt	14
average	28
rollups	88.94
rollups (sql)	8.33

Table 6.1: Execution Time

CHAPTER 7

Conclusion

The new wave of applications such as data warehousing, OLAPs and data mining have given rise to a need of *extensible, user-defined aggregates*. SQL-AG has been developed to support user-defined aggregates. It makes extensive use of complex aggregates by these applications feasible. SQL-AG is a simple but significantly more powerful extension as compared to previous extensions such as proposal of user-defined aggregates by SQL3 and POSTGRES. The later extensions have overlooked the need of *early returns* which is an essential component to support *online aggregation*.

SQL-AG is a system with simple specification language, it is easy to use and provides much more expressive power than SQL. Since SQL-AG uses PL/SQL for implementation it is closely integrated with SQL.

Performance

Though SQL-AG system has been implemented with due care to performance, it is lot slower than SQL. This can be attributed to the implementation environment used. Since SQL doesn't provide any way to directly extend its functionality, procedural extensions such as PL/SQL, Pro*C or other procedural languages have to be used to provide the functionality of SQL-AG. Thus the interaction between SQL-Procedural extension is the main overhead for any SQL extension. For SQL-

AG this overhead is minimum as compared to any implementation using language other than PL/SQL and extending SQL over Oracle. A direct integration of SQL-AG in database will drastically improve its performance.

Future Work

Implement practical applications in the area of OLAP, data mining and data warehousing using SQL-AG. The Apriori algorithm and Bayesian classifiers application are just a few of the applications for which SQL-AG could be used. Such use will define the utility of the system, along with a test of its performance. SQL-AG can be extended by providing support of SQL-AG on other systems. Rather than using PL/SQL, possibilities of using some other language like JDBC that can provide fast computation and close integration to database should be studied. Performance of SQL-AG can be improved by having an implementation using a closer integration to SQL engine. Use of application specific optimizations and the role of user-defined aggregates in extensible databases also deserve further investigation.

APPENDIX A

Example SQL-AG program

A.1 Maxname

```
aggregate maxname;
    Outputdefn output(v_result number, name varchar2(10));
    type internalrecord is record (
        maximum number,
        idused number);
    table_of_name t_stringtable;
    procedure single (v_f1 number, v_f2 varchar2(10),
        newvar OUT internalrecord) is
    begin
        newvar.maximum:=v_f1;
        newvar.idused:=1;
        table_of_name(newvar.idused):=v_f2;
    end ;
    procedure multi (v_f1 number, v_f2 varchar2(10), oldvar internalrecord,
        newvar IN OUT internalrecord) is
    begin
        if (oldvar.maximum ¡ v_f1) then
```

```
        newvar.maximum:=v_f1;
        newvar.idused:=1;
        table_of_name(newvar.idused):=v_f2;
    elsif (oldvar.maximum = v_f1) then
        newvar.maximum:=oldvar.maximum;
        newvar.idused:=oldvar.idused+1;
        table_of_name(newvar.idused):=v_f2;
    else
        newvar.maximum:=oldvar.maximum;
        newvar.idused:=oldvar.idused;
    end if;
end ;
procedure produce (oldvar internalrecord, flag varchar2) is
    i number;
begin
    if ( flag = 'normal_flag') then
        null;
    elsif (flag = 'eof_flag') then
        for i in 1..oldvar.idused loop
            Output(oldvar.maximum,table_of_name(i));
        end loop;
    end if;
end ;
```

.

A.2 Datacube

```
aggregate DataCube;
    Outputdefn output(year number, season varchar2(10), month varchar2(5), day number,
        yearTotal number, seasonTotal number, monthTotal number, dayTotal number);
    type internalrecord is record (
        t_year number,
        t_season number,
        t_month number,
        t_day number,
        c_year number,
        c_season varchar2(10),
        c_month varchar2(5),
        c_day number);
    procedure single (v_f1 number, v_f2 varchar2(10), v_f3 varchar2(5), v_f4 number,
        v_f5 number, newvar OUT internalrecord) is
    begin
        newvar.t_year:=0;
        newvar.t_season:=0;
        newvar.t_month:=0;
        newvar.t_day:=v_f5;
        newvar.c_year:=v_f1;
        newvar.c_season:=v_f2;
        newvar.c_month:=v_f3;
        newvar.c_day:=v_f4;
    end ;
```

```
procedure multi (v_f1 number, v_f2 varchar2(10), v_f3 varchar2(5), v_f4 number,
    v_f5 number, oldvar internalrecord, newvar IN OUT internalrecord) is
begin
    if (oldvar.c_year!=v_f1) then
        newvar.t_day:=oldvar.t_day;
        newvar.t_month:=oldvar.t_month + newvar.t_day;
        newvar.t_season := oldvar.t_season + newvar.t_month;
        newvar.t_year := oldvar.t_year + newvar.t_season;
        Output(oldvar.c_year, oldvar.c_season, oldvar.c_month, oldvar.c_day,
            newvar.t_year, newvar.t_season, newvar.t_month, newvar.t_day);
        newvar.t_year:=0;
        newvar.t_season:=0;
        newvar.t_month:=0;
        newvar.t_day:=v_f5;
        newvar.c_year:=v_f1;
        newvar.c_season:=v_f2;
        newvar.c_month:=v_f3;
        newvar.c_day:=v_f4;
    elsif (oldvar.c_season!=v_f2) then
        newvar.t_day:=oldvar.t_day;
        newvar.t_month:=oldvar.t_month + newvar.t_day;
        newvar.t_season := oldvar.t_season + newvar.t_month;
        newvar.t_year := oldvar.t_year + newvar.t_season;
        Output(oldvar.c_year, oldvar.c_season, oldvar.c_month, oldvar.c_day,
            0, newvar.t_season, newvar.t_month, newvar.t_day);
        newvar.t_season:=0;
```

```
    newvar.t_month:=0;
    newvar.t_day:=v_f5;
    newvar.c_year:=v_f1;
    newvar.c_season:=v_f2;
    newvar.c_month:=v_f3;
    newvar.c_day:=v_f4;
elsif (oldvar.c_month!=v_f3) then
    newvar.t_day:=oldvar.t_day;
    newvar.t_month:=oldvar.t_month + newvar.t_day;
    newvar.t_season := oldvar.t_season + newvar.t_month;
    newvar.t_year := oldvar.t_year;
    Output(oldvar.c_year, oldvar.c_season, oldvar.c_month, oldvar.c_day,
        0, 0, newvar.t_month, newvar.t_day);
    newvar.t_month:=0;
    newvar.t_day:=v_f5;
    newvar.c_year:=v_f1;
    newvar.c_season:=v_f2;
    newvar.c_month:=v_f3;
    newvar.c_day:=v_f4;
elsif (oldvar.c_day!=v_f4) then
    newvar.t_day := oldvar.t_day;
    newvar.t_month:=oldvar.t_month + newvar.t_day;
    newvar.t_season := oldvar.t_season ;
    newvar.t_year := oldvar.t_year ;
    Output(oldvar.c_year, oldvar.c_season, oldvar.c_month, oldvar.c_day,
        0, 0, 0, newvar.t_day);
```

```
        newvar.t_day:=v_f5;
        newvar.c_year:=v_f1;
        newvar.c_season:=v_f2;
        newvar.c_month:=v_f3;
        newvar.c_day:=v_f4;
    else
        newvar.t_year:=oldvar.t_year;
        newvar.t_season:=oldvar.t_season;
        newvar.t_month:=oldvar.t_month;
        newvar.t_day:=oldvar.t_day+v_f5;
        newvar.c_year:=v_f1;
        newvar.c_season:=v_f2;
        newvar.c_month:=v_f3;
        newvar.c_day:=v_f4;
    end if;
end ;
procedure produce (oldvar internalrecord, flag varchar2) is
    day number;
    month number;
    season number;
    year number;
begin
    if ( flag = 'normal_flag') then
        null;
    elsif (flag = 'eof_flag') then
        day:=oldvar.t_day;
```

```
        month:=oldvar.t_month + day;
        season := oldvar.t_season + month;
        year := oldvar.t_year + season;
        Output(oldvar.c_year, oldvar.c_season, oldvar.c_month, oldvar.c_day,
            year, season, month, day);
    end if;
end ;
```

APPENDIX B

Using Average

Following are calls to *average* aggregate on *sqlplus* command line, along with the output.

```
SQL> begin average('emp','sal', 'deptno');end;
  2   /
Create table InternalTable (DEPTNO number, Id number)
select deptno from InternalTable where Id = :g_id
select deptno,sal from emp
30 1940
20 3015
30 1775
10 2916.66666666666666666666666666666666667

PL/SQL procedure successfully completed.

SQL> begin average('emp','sal', 'deptno', 'temptable');end;
  2   /
Create table temptable (DEPTNO number, Id number, Result1 number)
Create table InternalTable (DEPTNO number, Id number)
select deptno from InternalTable where Id = :g_id
```

```
select deptno,sal from emp

PL/SQL procedure successfully completed.

SQL> select * from temptable;

DEPTNO ID RESULT1
———- ———- ———-
30 1 1940
20 2 3015
30 3 1775
10 4 2916.66667

SQL> begin average('emp','sal', 'deptno', 'temptable','OrderBy');end;
  2    /
Create table temptable (DEPTNO number, Id number, Result1 number)
Create table InternalTable (DEPTNO number, Id number)
select deptno from InternalTable where Id = :g_id
select deptno,sal from emp order by deptno

PL/SQL procedure successfully completed.

SQL> select * from temptable;

DEPTNO ID RESULT1
———- ———- ———-
```

10 1 2916.66667

20 2 3015

30 3 1880

30 4 1775

References

[AAD96] Sarneet Agarwal, Rakesh Agrawal, Prasad Deshpande, Ashish Gupta, Jeffrey F. Naughton, Raghu Ramakrishnan, and Sunita Sarawagi. "On the Computation of Multidimensional Aggregates." In *Proceedings of the 22nd VLDB Conference*, Bombay, India, 1996.

[AC97] Michael Abbey and Michael J. Corey. *A Beginner's Guide*. McGrawHill Publishing Company, 1997.

[AS94] Rakesh Agrawal and Ramkrishnan Srikant. "Fast Algorithm for Mininig Association Rules." In *Proceedings of the 20th VLDB Conference*, Santiago, Chile, 1994.

[DD93] C. J. Date and H. Darwen. *A Guide to SQL Standard.* Addison Wesley Publishing Company, 1993.

[Elk97] Charles Elkan. "Boosting and Naive Bayesian Learning." In *Proceedings of the Third Internation Conference on Knowldge Discovery and Data*, 1997.

[GBA96] Jim Gray, Adam Bosworth, and Hamid Pirahesh Andrew Layman. "Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Total." *International Conference on Data Engineering*, 1996.

[HHW97] J. M. Hellerstein, P. J. Haas, and H. J. Wang. "Online Aggregation." In *Proceedings of ACM-SIGMOD Conference on Management of Data*, May 1997.

[HRU96] Venky Harinarayan, Anand Rajaraman, and Jeffrey D. Ullman. "Implementing Data Cubes Efficiently." *In Proceedings of ACM-SIGMOD Conference on Management of Data*, 1996.

[Iso96a] ISO/IEC JTC1/SC21 N10489 ISO//IEC 9075 Part 2 Committee. "Database Language SQL - Part 2: SQL/Foundation." *ftp://speckle.ncsl.nist.gov/isowg3/dbl/BASEdocs/cd-found.pdf*, July 1996.

[Iso96b] ISO/IEC JTC1/SC21 N10491 and ISO//IEC 9075 Part 8 Committee. "Database Language SQL - Part 8: SQL/Object." *ftp://speckle.ncsl.nist.gov/isowg3/dbl/BASEdocs/cd-object.pdf*, July 1996.

[Mat96] Nelson Mattos. "An Overview of the SQL3 Standard." presentation foils, Database Technology Institute, IBM Santa Teresa Lab, San Jose,CA, ftp://speckle.ncsl.nist.gov/isowg3/dbl/BASEdocs/descriptions/SQL3_foils.ps, July 1996.

[MPC96] R. Meo, G. Psaila, and S. Ceri. "A New SQL-like Operator for Mining Association Rules." In *Proceedings of 22nd VLDB Conference*, Bombay, India, 1996.

[NT89] S. Naqvi and S. Tsur. *A Logical Language For Data and Knowledge Bases.* Computer Science Press, 1989.

[Red96] Red Brick Inc. "Decision-Makers, Business Data and RISQL." *http://www.redbrick.com/rbs-g/html/whpap.html*, August 1996.

[RS87] L. Rowe and M. Stonebraker. "The POSTGRES Data Model." In *Proceedings of VLDB Conference*, Brighton, England, September 1987.

[RS97] Kenneth A. Ross and Divesh Srivastava. "Fast Computation of Sparse Datacubes." In *Proceedings of the 23rd VLDB Conference*, Athens, Greece, 1997.

[SR86] M. Stonebraker and L. Rowe. "The Design of POSTGRES." In *Proceedings of ACM-SIGMOD Conference on Management of Data*, Washington, DC, May 1986.

[SRH90] M. Stonebraker, L. Rowe, and M. Hirohama. "The Implementation of POSTGRES." *IEEE Transactions on Knowledge and Data Engineering*, **2**(1), March 1990.

[Urm97] Scott Urman. *Oracle8 PL/SQL Programming.* McGrawHill Publishing Company, 1997.

[Wei98a] Wei-Min Shen. "Data Classification by Statistical Methods." *UCLA Data Mining Short Course(2) Notes*, 1998.

[Wei98b] Wei-Min Shen. "Data Classification for Data Mining." *UCLA Data Mining Short Course(1) Notes*, 1998.

[Wid95] J. Widom. "Research Problems in Data Warehousing." In *Proceeding of 4th International Conference on Information and Knowledge Management*, November 1995.

[Zan88] Carlo Zaniolo. "Design and Implementation of a Logic Based Language for Data Intensive Applications." In *Proceeding of the International Conference on Logic Programming*, 1988.

[ZAO93] Carlo Zaniolo, N. Arni, and K. Ong. "Negation and Aggregates in Recursive Rules: the LDL++ Approach." *Deductive Object Oriented Databases*, 1993.

[ZDN97] Yihong Zhao, Prasad Deshpande, and Jeffrey F. Naughton. "An Array-Based Algorithm for Simultaneous Multidimensional Aggregates." *In Proceedings of ACM-SIGMOD Conference on Management of Data*, 1997.

[ZM97] Carlo Zaniolo and I. Motakis. "Temporal Aggregation in Active Database Rules." *In Proceedings of ACM-SIGMOD Conference on Management of Data*, 1997.