SQL: Part 3

WITH clause - common table expr.

- WITH clause is an alternative to using derived tables
- WITH <tablename>[(column_name_list)] AS (<query expression>)

```
with item_avg_price(item, avg_price) as
      (select item, avg(price)
       from sp
       group by item)
select name, sp.item, price, avg_price
from sp, item_avg_price
where price <= avg_price and sp.item = item_avg_price.item
order by 1,2,3;
 name | item | price | avg_price
------+------+-------+----------
 S1   | P1   |    10 |   10.0
 S1   | P2   |    20 |   20.0
 S2   | P3   |   100 |  100.0
 S3   | P4   |  1000 | 1000.0
 S4   | P1   |     9 |   10.0
(5 rows)
```

WITH vs. Derived Table

- Use the one that seems to make more sense in developing the query

```
select name, sp.item, price, avg_price
from sp, (select item, avg(price)
          from sp
          group by item) item_avg_price(item, avg_price)
where price <= avg_price and sp.item = item_avg_price.item
order by 1,2,3;
```

```
with item_avg_price(item, avg_price) as
      (select item, avg(price)
       from sp
       group by item)
select name, sp.item, price, avg_price
from sp, item_avg_price
where price <= avg_price and sp.item = item_avg_price.item
order by 1,2,3;
```

Recursion and WITH

- WITH clause can be used for simple tail recursive expressions
- Useful for exploring hierarchical data
- Find all spaces in the CS building

```
with recursive csspaces(sp, pa, area) as(
   select space, parent, sqft
   from spaces
   where space = 'Comp Sci Bldg'
   union all
   select space, parent, sqft
   from csspaces, spaces
   where csspaces.sp  = spaces.parent
   )
select sp, pa, area
from csspaces;
        sp           |          pa           | area
---------------------+-----------------------+------
 Comp Sci Bldg       | UW Campus             |
 CS 3rd Wing         | Comp Sci Bldg         |
 CS 2nd Wing         | Comp Sci Bldg         |
 CS 4th Flr 3rd Wing | CS 3rd Wing           |
 CS 1st Flr 2nd Wing | CS 2nd Wing           |
 CS 5th Flr 3rd Wing | CS 3rd Wing           |
 Rm4369              | CS 4th Flr 3rd Wing   |  100
 Rm4361              | CS 4th Flr 3rd Wing   |  120
 Rm1240              | CS 1st Flr 2nd Wing   | 1000
 Rm5310              | CS 5th Flr 3rd Wing   |  200
```

Recursive Execution

- Result table (csspaces) and Temp Table (TT)
- both start with Comp Sci Bldg
- TT joined with spaces table adding
  - 2 wings to result table
  - creating new TT with 2 wings
- new TT joined with spaces adding
  - 3 floors to result table
  - creating new TT with 3 floors
- new TT joined with spaces adding
  - 4 rooms
  - creating new TT with 4 rooms
- new TT joined with spaces adding nothing
  - DONE

space	parent	sqft
Rm4369	CS 4th Flr 3rd Wing	100
Rm4361	CS 4th Flr 3rd Wing	120
CS 4th Flr 3rd Wing	CS 3rd Wing	
CS 3rd Wing	Comp Sci Bldg	
Rm1240	CS 1st Flr 2nd Wing	1000
CS 1st Flr 2nd Wing	CS 2nd Wing	
CS 2nd Wing	Comp Sci Bldg	
Rm5310	CS 5th Flr 3rd Wing	200
CS 5th Flr 3rd Wing	CS 3rd Wing	
Comp Sci Bldg	UW Campus	
Edu Sci 240	Edu Sci 2nd Flr	500
Edu Sci 2nd Flr	Edu Sci Bldg	
Edu Sci Bldg	UW Campus	

(13 rows)

sp	pa	area
Comp Sci Bldg	UW Campus	
CS 3rd Wing	Comp Sci Bldg	
CS 2nd Wing	Comp Sci Bldg	
CS 4th Flr 3rd Wing	CS 3rd Wing	
CS 1st Flr 2nd Wing	CS 2nd Wing	
CS 5th Flr 3rd Wing	CS 3rd Wing	
Rm4369	CS 4th Flr 3rd Wing	100
Rm4361	CS 4th Flr 3rd Wing	120
Rm1240	CS 1st Flr 2nd Wing	1000
Rm5310	CS 5th Flr 3rd Wing	200

Another Example

- $t = \{1\}$, $TT = \{1\}$
- $t = \{1\} \cup \{2\}$, $TT = \{2\}$
- $t = \{1,2\} \cup \{3\}$, $TT = \{3\}$
- $t = \{1,2,3\} \cup \{4\}$, $TT = \{4\}$…
- $t = \{1,2,3,4,5,6,7,8\} \cup \{9\}$, $TT = \{9\}$
- $t = \{1, \ldots, 9\} \cup \{10\}$, $TT = \{10\}$
- TT in next iteration adds no rows
- final $t = \{1, \ldots, 10\}$

```
with recursive t(n) AS (
  select 1
  union all
  select n+1 from t where n < 10
)
select * from t;
 n
----
  1
  2
  3
  4
  5
  6
  7
  8
  9
 10
(10 rows)
```

Review

WITH <with clauses>
SELECT [options] column_expression_list
FROM table_expression_list
WHERE condition
GROUP BY groupby_list/ordinal_list
HAVING condition
ORDER BY column_expression_list/ordinal_list

- Conceptually, we first evaluate the cross-products, joins and WHERE conditions
- If there is a (GROUP BY or Aggregate Functions in SELECT/ORDER BY)
  - Then we aggregate the rows according to the GROUP BY expression computing functions mentioned in SELECT list, ORDER BY list and HAVING condition
  - Then we apply the HAVING condition, if any, to the resulting rows
- If there are Ordered-analytic functions, we evaluate them
- Then evaluate the SELECT list column expressions and the ORDER BY expressions
- Finally, the output rows are sorted in according to the ORDER BY

FROM clause

- Tables, Views, Common Table Expressions (WITH), Derived Tables, and Joined Tables
- Derived Tables =
(<query expression>) [AS] <name>(column_name_list)
  - must have unique names for all columns
- Joined Tables, e.g. T1 left outer join T2 ON <condition>
- Specialized: Table Functions and Table Operators, PIVOT/UNPIVOT etc.

Outer Joins

- join = inner join
- join as we know it

```
select *
from t1;
 a | b
---+----
 1 | 10
 2 | 20
 3 | 30
 4 | 40
(4 rows)
```

```
select *
from t2;
 a |  c
---+-----
 0 |   0
 1 | 100
 2 | 200
(3 rows)
```

```
select *
from t1 join t2
     on t1.a = t2.a;
 a | b  | a |  c
---+----+---+-----
 1 | 10 | 1 | 100
 2 | 20 | 2 | 200
(2 rows)
```

Outer Joins

```
select *
from t1;
 a | b
---+----
 1 | 10
 2 | 20
 3 | 30
 4 | 40
```

```
select *
from t2;
 a |  c
---+-----
 0 |    0
 1 | 100
 2 | 200
```

```
select * from t1 left outer join t2 on t1.a = t2.a;
 a | b  | a |  c
---+----+---+-----
 1 | 10 | 1 | 100
 2 | 20 | 2 | 200
 3 | 30 |   |
 4 | 40 |   |
```

- T1 left outer join T2
  - rows of T1 that don't join with T2 are included with NULL values for the T2 columns

```
select * from t1 right join t2 on t1.a = t2.a;
 a | b  | a |  c
---+----+---+-----
   |    | 0 |    0
 1 | 10 | 1 | 100
 2 | 20 | 2 | 200
```

- T1 right outer join T2
  - symmetric
- T1 full outer join T2
  - keep rows of both T1 & T2

```
select * from t1 full outer join t2 on t1.a = t2.a;
 a | b  | a |  c
---+----+---+-----
   |    | 0 |    0
 1 | 10 | 1 | 100
 2 | 20 | 2 | 200
 3 | 30 |   |
 4 | 40 |   |
```

WHERE clause

- Any boolean condition, using row values generated by the FROM clause
- Special functions available for strings, NULLs, etc.
- Subqueries (uncorrelated and correlated)

GROUP BY clause

- GROUP BY clause => aggregation
- aggregation functions in SELECT, HAVING, ORDER BY
- No aggregations in WHERE - why?
- GROUP BY 1, 3, 5 would GROUP BY on 1st, 3rd, 5th column of the SELECT list (a handy short-cut)

HAVING clause

- Any boolean condition, using row values generated by the GROUP BY aggregation step
- Special functions available for strings, NULLs, etc.
- Subqueries (uncorrelated and correlated)

Ordered-Analytic Functions

- Presence of Ordered-Analytic (window) functions in SELECT or ORDER BY implies the additional step

SELECT clause

- SELECT list can have any expressions that are based on the row values “generated” so far (depending on FROM, WHERE, GROUP BY)
- DISTINCT option, also TOP N option (combined with ORDER BY)

ORDER BY clause

- ORDER BY clause allowed in the outer-most SELECT query expression (or their UNION, etc.) as relations/multi-sets are un-ordered (exception: ORDER BY combined with TOP N).
- ORDER BY can include values not in the SELECT list
- ORDER BY 1, 3, 5 would ORDER BY on 1st, 3rd, 5th column of the SELECT list (a handy short-cut)
- NULLs can be put first/last (or default - system dependent).
- ORDER BY 1 asc, 2 desc, 3 asc nulls first
    - asc is default

ORDER BY example

```
select * from t3;
 a | b
---+----
 1 | 10
 1 | 11
 1 | 12
   | 10
   | 11
 2 |
 2 | 11
 2 | 12
(8 rows)
```

```
select * from t3
order by a asc nulls first, b desc nulls last;
 a | b
---+----
   | 11
   | 10
 1 | 12
 1 | 11
 1 | 10
 2 | 12
 2 | 11
 2 |
(8 rows)
```

Case Study

- spaces table defines a hierarchy with floor space only given at the leaf nodes
- Find the aggregated floor space value at each level of the hierarchy for the Comp Sci Bldg

```
select * from spaces;
        space          |        parent         | sqft
-----------------------+-----------------------+------
 Rm4369                | CS 4th Flr 3rd Wing   |  100
 Rm4361                | CS 4th Flr 3rd Wing   |  120
 CS 4th Flr 3rd Wing   | CS 3rd Wing           |
 CS 3rd Wing           | Comp Sci Bldg         |
 Rm1240                | CS 1st Flr 2nd Wing   | 1000
 CS 1st Flr 2nd Wing   | CS 2nd Wing           |
 CS 2nd Wing           | Comp Sci Bldg         |
 Rm5310                | CS 5th Flr 3rd Wing   |  200
 CS 5th Flr 3rd Wing   | CS 3rd Wing           |
 Comp Sci Bldg         | UW Campus             |
 Edu Sci 240           | Edu Sci 2nd Flr       |  500
 Edu Sci 2nd Flr       | Edu Sci Bldg          |
 Edu Sci Bldg          | UW Campus             |
(13 rows)
```

Step 1

- Find the rooms for the CS building …

```
with recursive csspaces(sp, pa, area) as(
     select space, parent, sqft from spaces where space = 'Comp Sci Bldg
     union all
     select space, parent, sqft
     from csspaces, spaces
     where csspaces.sp  = spaces.parent
     )
select sp, pa, area
from csspaces
where area is not null;
   sp   |         pa           | area
--------+----------------------+------
 Rm4369 | CS 4th Flr 3rd Wing |  100
 Rm4361 | CS 4th Flr 3rd Wing |  120
 Rm1240 | CS 1st Flr 2nd Wing | 1000
 Rm5310 | CS 5th Flr 3rd Wing |  200
(4 rows)
```

Step 2

- Push the area value to the "parent" level

```
with recursive newcsspaces(sp, pa, area) as
(
    select sp, pa, area
    from (
    with recursive csspaces(sp, pa, area)
     as(
       select space, parent, sqft
       from spaces
       where space = 'Comp Sci Bldg'
       union all
       select space, parent, sqft
       from csspaces, spaces
       where csspaces.sp  = spaces.parent
       )
    select sp, pa, area
    from csspaces
    where area is not null
    ) csrms
    union all
    select space, parent,
                 area + coalesce(sqft, 0)
    from newcsspaces, spaces
    where newcsspaces.pa = spaces.space
    )
select * from newcsspaces;
```

```
         sp           |         pa         | area
----------------------+--------------------+-----
 Rm4369               | CS 4th Flr 3rd Wing|  100
 Rm4361               | CS 4th Flr 3rd Wing|  120
 Rm1240               | CS 1st Flr 2nd Wing| 1000
 Rm5310               | CS 5th Flr 3rd Wing|  200
 CS 4th Flr 3rd Wing  | CS 3rd Wing        |  100
 CS 4th Flr 3rd Wing  | CS 3rd Wing        |  120
 CS 1st Flr 2nd Wing  | CS 2nd Wing        | 1000
 CS 5th Flr 3rd Wing  | CS 3rd Wing        |  200
 CS 3rd Wing          | Comp Sci Bldg      |  100
 CS 3rd Wing          | Comp Sci Bldg      |  120
 CS 2nd Wing          | Comp Sci Bldg      | 1000
 CS 3rd Wing          | Comp Sci Bldg      |  200
 Comp Sci Bldg        | UW Campus          |  100
 Comp Sci Bldg        | UW Campus          |  120
 Comp Sci Bldg        | UW Campus          | 1000
 Comp Sci Bldg        | UW Campus          |  200
(16 rows)
```

Aggregate

- But something is missing

```
with newcsspaces (
    …
)
select sp, pa,sum(area)
from newcsspaces
group by sp, pa
order by 3;
         sp          |          pa          | sum
---------------------+----------------------+------
 Rm4369              | CS 4th Flr 3rd Wing  |  100
 Rm4361              | CS 4th Flr 3rd Wing  |  120
 CS 5th Flr 3rd Wing | CS 3rd Wing          |  200
 Rm5310              | CS 5th Flr 3rd Wing  |  200
 CS 4th Flr 3rd Wing | CS 3rd Wing          |  220
 CS 3rd Wing         | Comp Sci Bldg        |  420
 CS 1st Flr 2nd Wing | CS 2nd Wing          | 1000
 Rm1240              | CS 1st Flr 2nd Wing  | 1000
 CS 2nd Wing         | Comp Sci Bldg        | 1000
 Comp Sci Bldg       | UW Campus            | 1420
(10 rows)
```

Adding “level” value

```
with recursive newcsspaces(sp, pa, area, level) as (
    select sp, pa, area, 1
    from (
    with recursive csspaces(sp, pa, area) as(
       select space, parent, sqft from spaces where space = 'Comp Sci Bldg'
       union all
       select space, parent, sqft
       from csspaces, spaces
       where csspaces.sp  = spaces.parent
       )
    select sp, pa, area
    from csspaces
    where area is not null
    ) csrms
    union all
    select space, parent, area + coalesce(sqft, 0), level + 1
    from newcsspaces, spaces
    where newcsspaces.pa = spaces.space
    )
select sp, pa,sum(area)
from newcsspaces
group by sp, pa, level
order by level, 3;
```

Result

sp	pa	sum
Rm4369	CS 4th Flr 3rd Wing	100
Rm4361	CS 4th Flr 3rd Wing	120
Rm5310	CS 5th Flr 3rd Wing	200
Rm1240	CS 1st Flr 2nd Wing	1000
CS 5th Flr 3rd Wing	CS 3rd Wing	200
CS 4th Flr 3rd Wing	CS 3rd Wing	220
CS 1st Flr 2nd Wing	CS 2nd Wing	1000
CS 3rd Wing	Comp Sci Bldg	420
CS 2nd Wing	Comp Sci Bldg	1000
Comp Sci Bldg	UW Campus	1420

(10 rows)

Another Way

```
with csrooms as(
    with recursive csspaces(sp, pa, area) as(
       select space, parent, sqft from spaces where space = 'Comp Sci Bldg'
       union all
       select space, parent, sqft
       from csspaces, spaces
       where csspaces.sp  = spaces.parent
       )
    select sp, pa, area
    from csspaces
    where area is not null)
select bldg.space as bldg, floor.space as floor, wing.space as wing,
       csrooms.sp as room, sum(area)
from csrooms, spaces floor, spaces wing, spaces bldg
where csrooms.pa = floor.space and floor.parent = wing.space and
      wing.parent = bldg.space
group by rollup(1, 2, 3, 4);
```

bldg	floor	wing	room	sum
Comp Sci Bldg	CS 1st Flr 2nd Wing	CS 2nd Wing	Rm1240	1000
Comp Sci Bldg	CS 1st Flr 2nd Wing	CS 2nd Wing		1000
Comp Sci Bldg	CS 1st Flr 2nd Wing			1000
Comp Sci Bldg	CS 4th Flr 3rd Wing	CS 3rd Wing	Rm4361	120
Comp Sci Bldg	CS 4th Flr 3rd Wing	CS 3rd Wing	Rm4369	100
Comp Sci Bldg	CS 4th Flr 3rd Wing	CS 3rd Wing		220
Comp Sci Bldg	CS 4th Flr 3rd Wing			220
Comp Sci Bldg	CS 5th Flr 3rd Wing	CS 3rd Wing	Rm5310	200
Comp Sci Bldg	CS 5th Flr 3rd Wing	CS 3rd Wing		200
Comp Sci Bldg	CS 5th Flr 3rd Wing			200
Comp Sci Bldg				1420
				1420

(12 rows)

Traversing Hierarchy, Self Join

- Self Join = when one joins the table to itself
  - different role for the same table
- Useful in traversing fixed level hierarchies
- Must “alias” table names in FROM clause

Database Extensibility

- To extend / customize the database system
- User Defined Functions
  - scalar functions
  - aggregate function
  - some systems allow user defined window functions
- User Defined Types
- Table Functions and Table Operators

History of Extensibility

- New DBMS variations (e.g. OO DBMS etc.) to overcome the “fixed” nature of the old RDBMS
- But then RDBMS made extensible in response
- Make it easy to handle new data types and functionality
  - match the essential SQL semantics and flow
- Vendors too can leverage it for quicker rollout

Scalar UDF's

- Scalar means the "good old kind": value = f(x,y,z,..)
- Add a function that's not available, say trig functions
- Something custom that's not easily expressible in SQL: e.g. SOUNDEX
- Can also be simply "syntactic sugar"
- Written in C, Java, SQL, custom languages (pgsql)

Scalar UDF example

```
db1=# create function incr(val integer)
db1-# returns integer as $$
db1$# begin
db1$#   return val + 1;
db1$# end; $$
db1-# language plpgsql;
CREATE FUNCTION
db1=# select a, incr(a) from t1;
 a | incr
---+------
 1 |    2
 2 |    3
 3 |    4
 4 |    5
(4 rows)
```

```
db1=# create function add_one(integer)
db1-# returns integer
db1-#   as '/Users/ambuj/postgres/udf',
db1-#      'add_one'
db1-# language C strict;
CREATE FUNCTION
db1=# select a, incr(a), add_one(a)
db1-# from t1;
 a | incr | add_one
---+------+---------
 1 |    2 |       2
 2 |    3 |       3
 3 |    4 |       4
 4 |    5 |       5
(4 rows)
```

```
#include "postgres.h"
#include "fmgr.h"
#include <string.h>
#ifdef PG_MODULE_MAGIC
PG_MODULE_MAGIC;
#endif

int add_one (int arg)
{
return arg + 1;
}
```

- Many ways to define a function
  - pgsql
  - C

UDT's

- What if we want to store spatial data
- SQL doesn't provide spatial type by default
- Define the type and functions that work on it and then use it as regular type

Aggregate UDF's

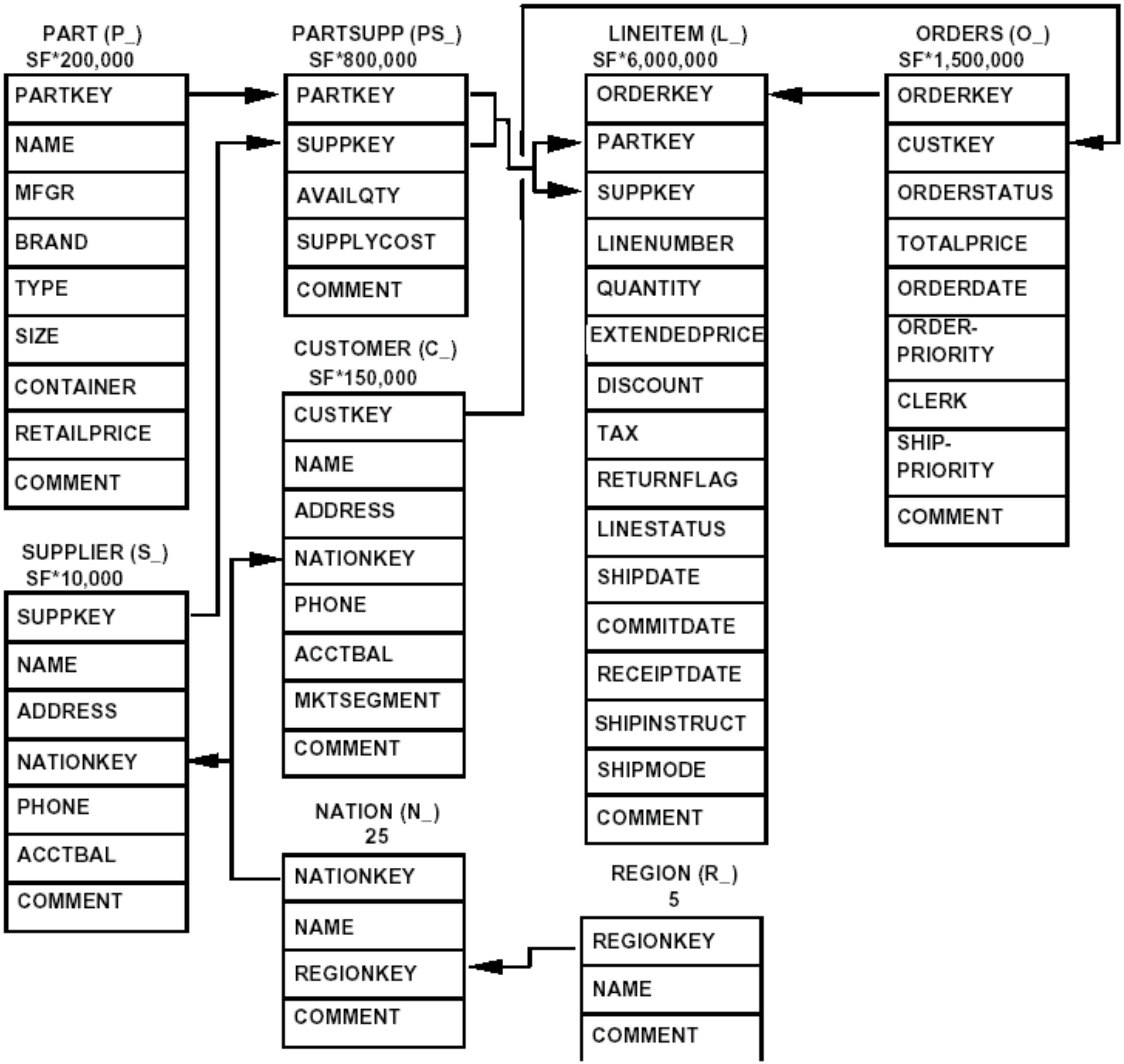
- Some systems allow one to define class of functions that would be used like aggregate functions
- have to define the details of the aggregation
  - system dependent

A Case Study: TPC-H Benchmark

- TPC is industry consortium for evaluating DBMS products
- TPC-H is a decision-support workload benchmark (as opposed to OLTP benchmark like TPC-C)
    - large data volume
        - updated frequently, but not at OLTP levels
    - complex queries representing business questions

TPC-H Schema

- A general business managing, selling, distributing worldwide
- Lineitem is the biggest table
- Data periodically refreshed

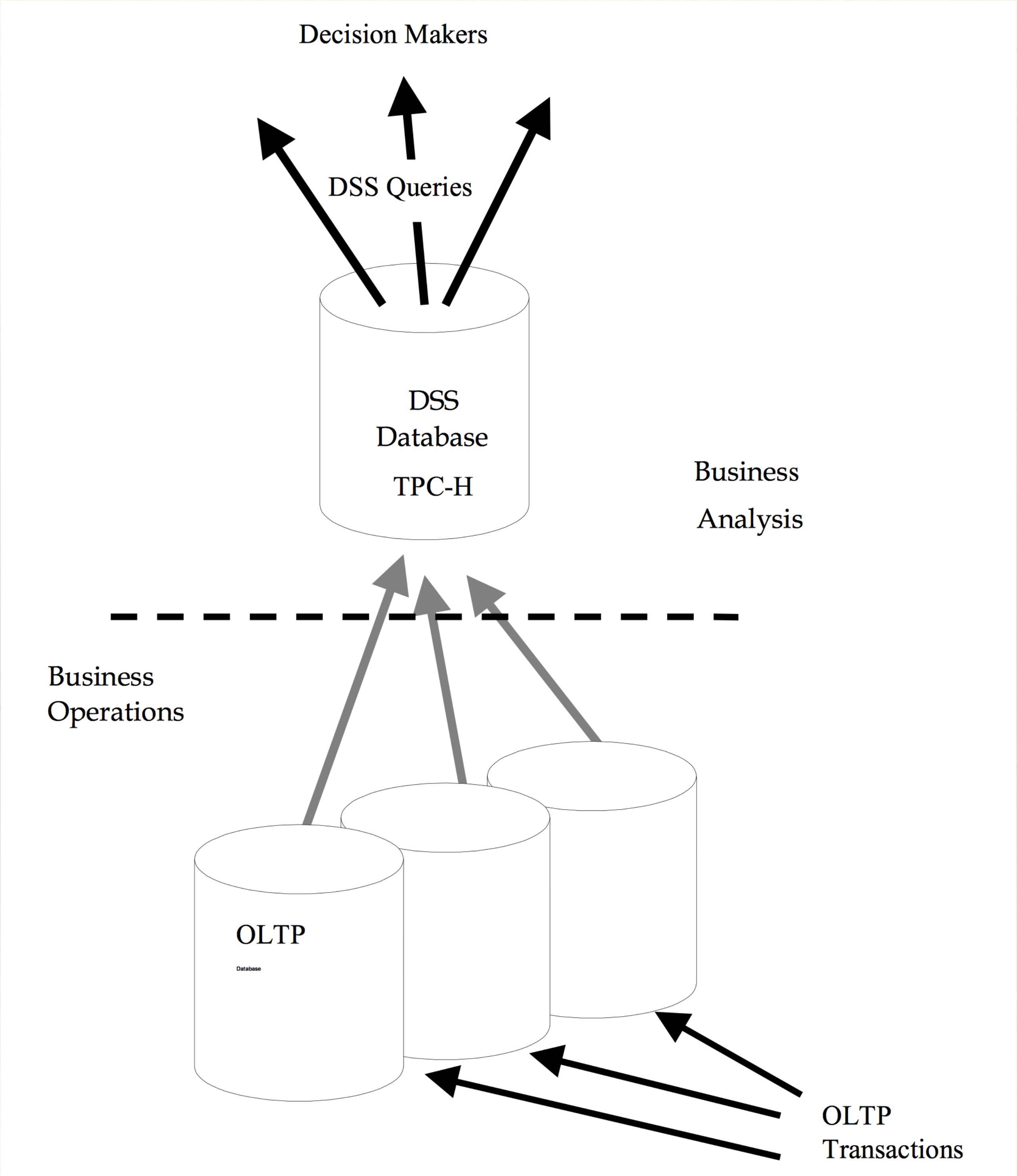


Legend:

- The parentheses following each table name contain the prefix of the column names for that table;
- The arrows point in the direction of the one-to-many relationships between tables;
- The number/formula below each table name represents the cardinality (number of rows) of the table. Some are factored by SF, the Scale Factor, to obtain the chosen database size. The cardinality for the LINEITEM table is approximate (see Clause 4.2.5).

A Simple DSS Model

- This is a very generic DSS model where data is fed from operational systems to the DSS system
- One can also have near-real time updates in a DSS (Teradata ADW is an example).



LineItem Table

- Do all terms look familiar?

LINEITEM Table Layout

Column Name	Datatype Requirements	Comment
L_ORDERKEY	identifier	Foreign Key to O_ORDERKEY
L_PARTKEY	identifier	Foreign key to P_PARTKEY, first part of the compound Foreign Key to (PS_PARTKEY, PS_SUPPKEY) with L_SUPPKEY
L_SUPPKEY	Identifier	Foreign key to S_SUPPKEY, second part of the compound Foreign Key to (PS_PARTKEY, PS_SUPPKEY) with L_PARTKEY
L_LINENUMBER	integer	
L_QUANTITY	decimal	
L_EXTENDEDPRICE	decimal	
L_DISCOUNT	decimal	
L_TAX	decimal	
L_RETURNFLAG	fixed text, size 1	
L_LINESTATUS	fixed text, size 1	
L_SHIPDATE	date	
L_COMMITDATE	date	
L_RECEIPTDATE	date	
L_SHIPINSTRUCT	fixed text, size 25	
L_SHIPMODE	fixed text, size 10	
L_COMMENT	variable text size 44	

Primary Key: L_ORDERKEY, L_LINENUMBER

Query 1: Amount of business

- The Pricing Summary Report Query provides a summary pricing report for all lineitems shipped as of a given date (varies). The query lists totals for various prices, and average discount, grouped by RETURNFLAG and LINESTATUS, and listed in ascending order of RETURNFLAG and LINESTATUS. A count of the number of lineitems in each group is included.

```
select
        l_returnflag,
        l_linestatus,
        sum(l_quantity) as sum_qty,
        sum(l_extendedprice) as sum_base_price,
        sum(l_extendedprice*(1-l_discount)) as sum_disc_price,
        sum(l_extendedprice*(1-l_discount)*(1+l_tax)) as sum_charge,
        avg(l_quantity) as avg_qty,
        avg(l_extendedprice) as avg_price,
        avg(l_discount) as avg_disc,
        count(*) as count_order
from
        lineitem
where
        l_shipdate <= date '1998-12-01' - interval '[DELTA]' day (3)
group by
        l_returnflag,
        l_linestatus
order by
        l_returnflag,
        l_linestatus;
```

Query 4: Check order priority system

- The Order Priority Checking Query counts the number of orders ordered in a given quarter of a given year in which at least one lineitem was received by the customer later than its committed date. The query lists the count of such orders for each order priority sorted in ascending priority order.

```
select
        o_orderpriority,
        count(*) as order_count
from
        orders
where
        o_orderdate >= date '[DATE]'
        and o_orderdate < date '[DATE]' + interval '3' month
        and exists (
                select
                        *
                from
                        lineitem
                where
                        l_orderkey = o_orderkey
                        and l_commitdate < l_receiptdate
        )
group by
        o_orderpriority
order by
        o_orderpriority;
```

Query 8: Market share of a nation

- The market share for a given nation within a given region is defined as the fraction of the revenue, the sum of [l_extendedprice * (1-l_discount)], from the products of a specified type in that region that was supplied by suppliers from the given nation. The query determines this for the years 1995 and 1996 presented in this order.
- SUM over a CASE
- nested aggregation

```
select
        o_year,
        sum(case
                when nation = '[NATION]'
                then volume
                else 0
        end) / sum(volume) as mkt_share
from (
        select
                extract(year from o_orderdate) as o_year,
                l_extendedprice * (1-l_discount) as volume,
                n2.n_name as nation
        from
                part,
                supplier,
                lineitem,
                orders,
                customer,
                nation n1,
                nation n2,
                region
        where
                p_partkey = l_partkey
                and s_suppkey = l_suppkey
                and l_orderkey = o_orderkey
                and o_custkey = c_custkey
                and c_nationkey = n1.n_nationkey
                and n1.n_regionkey = r_regionkey
                and r_name = '[REGION]'
                and s_nationkey = n2.n_nationkey
                and o_orderdate between date '1995-01-01' and date '1996-12-31'
                and p_type = '[TYPE]'
        ) as all_nations
group by
        o_year
order by
        o_year;
```

Query 10: Customers with problem parts

- The Returned Item Reporting Query finds the top 20 customers, in terms of their effect on lost revenue for a given quarter, who have returned parts. The query considers only parts that were ordered in the specified quarter. The query lists the customer's name, address, nation, phone number, account balance, comment information and revenue lost. The customers are listed in descending order of lost revenue. Revenue lost is defined as sum(l_extendedprice*(1-l_discount)) for all qualifying lineitems.
- SQL-92 didn't have TOP N option, so you fetched first 20 rows and quit
- Difference?

```
select TOP 20
        c_custkey,
        c_name,
        sum(l_extendedprice * (1 - l_discount)) as revenue,
        c_acctbal,
        n_name,
        c_address,
        c_phone,
        c_comment
from
        customer,
        orders,
        lineitem,
        nation
where
        c_custkey = o_custkey
        and l_orderkey = o_orderkey
        and o_orderdate >= date '[DATE]'
        and o_orderdate < date '[DATE]' + interval '3' month
        and l_returnflag = 'R'
        and c_nationkey = n_nationkey
group by
        c_custkey,
        c_name,
        c_acctbal,
        c_phone,
        n_name,
        c_address,
        c_comment
order by
        revenue desc;
```

Query 11: important subset of supplier stock

- The Important Stock Identification Query finds, from scanning the available stock of suppliers in a given nation, all the parts that represent a significant percentage of the total value of all available parts. The query displays the part number and the value of those parts in descending order of value.
- Subquery in HAVING

```
select
        ps_partkey,
        sum(ps_supplycost * ps_availqty) as value
from
        partsupp,
        supplier,
        nation
where
        ps_suppkey = s_suppkey
        and s_nationkey = n_nationkey
        and n_name = '[NATION]'
group by
        ps_partkey having
                sum(ps_supplycost * ps_availqty) > (
                        select
                                sum(ps_supplycost * ps_availqty) * [FRACTION
                        from
                                partsupp,
                                supplier,
                                nation
                        where
                                ps_suppkey = s_suppkey
                                and s_nationkey = n_nationkey
                                and n_name = '[NATION]'
                )
order by
        value desc;
```

Query 13: Customers and order size

- This query determines the distribution of customers by the number of orders they have ma including customers who have no record of orders, past or present. It counts and reports ho many customers have no orders, how many have 1, 2, 3, etc. A check is made to ensure that orders counted do not fall into one of several special categories of orders. Special categorie identified in the order comment column by looking for a particular pattern.

- notice the outer join to get all customers - even the non-ordering ones
- “like” matches the keywords being looked for special case
- derived table with column names

```
select
        c_count, count(*) as custdist
from (
        select
                c_custkey,
                count(o_orderkey)
        from
                customer left outer join orders on
                        c_custkey = o_custkey
                        and o_comment not like ‘%[WORD1]%[WORD2]%’
        group by
                c_custkey
        )as c_orders (c_custkey, c_count)
group by
        c_count
order by
        custdist desc,
        c_count desc;
```

Query 17: Revenue from small orders

- The Small-Quantity-Order Revenue Query considers parts of a given brand and with a given container type and determines the average lineitem quantity of such parts ordered for all orders (past and pending) in the 7-year data- base. What would be the average yearly gross (undiscounted) loss in revenue if orders for these parts with a quantity of less than 20% of this average were no longer taken?

```
select
        sum(l_extendedprice) / 7.0 as avg_yearly
from
        lineitem,
        part
where
        p_partkey = l_partkey
        and p_brand = '[BRAND]'
        and p_container = '[CONTAINER]'
        and l_quantity < (
                select
                        0.2 * avg(l_quantity)
                from
                        lineitem
                where
                        l_partkey = p_partkey
        );
```

TPC-H

- Very simple benchmark but contains the essentials of a business analysis workload
- TPC-DS is a newer, more complex, benchmark that uses additional, newer features, e.g. ROLLUP, RANK, with more complex schema
- Are they realistic?