Using a Database

Advanced SQL

SQL

- intergalactic standard for data retrieval
- SQL is declarative (specifies rather than how to execute the query)
- supported by almost all data processing platforms
- the SQL standard grows over time: SQL-92 (basics supported by most systems), SQL:1999 (recursive CTEs, grouping sets), SQL:2003 (window functions), SQL:2006 (XML), SQL:2008, SQL:2011 (temporal support, more window functions)
- newer versions of the SQL standard are huge and many systems only support subsets
- some differences in syntax and semantics between different systems
- we use the PostgreSQL dialect, which is fairly standards-compliant
TPC-H Data Set

- TPC-H is an ad-hoc, decision support benchmark
- randomly generated data set, data set generator available
- size can be configured (scale factor 1 is around 1 GB)
Loading Data: Schema

```
create table lineitem (  
l_orderkey integer not null, 
l_partkey integer not null, 
l_suppkey integer not null, 
l_linenumber integer not null, 
l_quantity decimal(12,2) not null, 
l_extendedprice decimal(12,2) not null, 
l_discount decimal(12,2) not null, 
l_tax decimal(12,2) not null, 
l_returnflag char(1) not null, 
l_linestatus char(1) not null, 
l_shipdate date not null, 
l_commitdate date not null, 
l_receiptdate date not null, 
l_shipinstruct char(25) not null, 
l_shipmode char(10) not null, 
l_comment text not null
);
```
Data Types (PostgreSQL)

- **signed integers**: smallint (2 bytes), integer (4 bytes), bigint (8 bytes)
- **fixed-precision numbers**: numeric(scale,precision) (scale is the number of decimal digits, precision is the number of digits after the decimal separator)
- **arbitrary precision numbers**: numeric (size unbounded, very slow)
- **floating point numbers**: float (4 bytes), double precision (8 bytes)
- **strings**: varchar(n) (maximum length n), char(n) (maximum length n, blank padded, strange semantics), text (arbitrary length)
- **other common types**: bytea (variable-length binary array), timestamp (8 bytes), date (4 bytes), interval (16 bytes), boolean (1 byte)
- all types may be NULL, unless NOT NULL is specified
- PostgreSQL stores data in row-wise fashion
- https://www.postgresql.org/docs/current/static/datatype.html
Using a Database

Advanced SQL

Loading CSV-like Data

$ head -n 1 lineitem.tbl
1|15519|785|1|17| ... |egular courts above the|
$ psql tpch

tpch=# \copy lineitem from lineitem.tbl delimiter ','
ERROR: extra data after last expected column
CONTEXT: COPY lineitem, line 1: "1|15519|785|1|17|24386.67|0.04|0.02|N|O|1996-03-13|1996-02-12|1996-03-22|DELIVER IN PERSON|TRUCK|egular courts above the|

tpch# \q
$ sed -i 's/|$//' lineitem.tbl
$ psql

$ psql

tpch=# \copy lineitem from lineitem.tbl delimiter ','
COPY 600572

- https://www.postgresql.org/docs/current/static/sql-copy.html
Basic SQL: Joins, Group By, Ordering

```sql
select l_orderkey, -- single line comment
    sum(l_extendedprice * (1 - l_discount)) revenue,
    o_orderdate, o_shippriority
from customer, orders, lineitem /* this is a
multi line comment */
where c_mktsegment = 'BUILDING'
and c_custkey = o_custkey
and l_orderkey = o_orderkey
and o_orderdate < date '1995-03-15'
and l_shipdate > date '1995-03-15'
group by l_orderkey, o_orderdate, o_shippriority
order by revenue desc, o_orderdate
limit 10;
```
• How many items were shipped by German suppliers in 1995?
• How many items were shipped by German suppliers in 1995?
• What are the names and account balances of the 10 customers from EUROPE in the FURNITURE market segment who have the highest account balance?
Subqueries

- subqueries can appear (almost) everywhere:

```sql
select n_name,
    (select count(*) from region)
from nation,
    (select *
    from region
    where r_name = 'EUROPE') region
where n_regionkey = r_regionkey
and exists (select 1
    from customer
    where n_nationkey = c_nationkey);
```
Correlated Subqueries

```sql
select avg(l_extendedprice)
from lineitem l1
where l_extendedprice =
    (select min(l_extendedprice)
     from lineitem l2
     where l1.l_orderkey = l2.l_orderkey);
```

- Subquery is correlated if it refers to tuple from outer query (l1.l_orderkey)
- Naive execution: execute inner query for every tuple of outer query (quadratic runtime unless index on l_orderkey exists)
Query Decorrelation

- queries can be rewritten to avoid correlation (some systems do this automatically):

```sql
select avg(l_extendedprice)
from lineitem l1,
     (select min(l_extendedprice) m, l_orderkey
      from lineitem
      group by l_orderkey) l2
where l1.l_orderkey = l2.l_orderkey
and l_extendedprice = m;
```
• decorrelate the following query

```sql
select c1.c_name
from customer c1
where c1.c_mktsegment = 'AUTOMOBILE'
or c1.c_acctbal >
  (select avg(c2.c_acctbal)
   from customer c2
   where c2.c_mktsegment = c1.c_mktsegment);
```
Set Operators

- UNION, EXCEPT, and INTERSECT remove duplicates

```sql
select n_name from nation where n_regionkey = 2
union
select n_name from nation where n_regionkey in (1, 2)
intersect
select n_name from nation where n_regionkey < 3
except
select n_name from nation where n_nationkey = 21;
```
“Set” Operators

- **UNION ALL**: $l + r$
- **EXCEPT ALL**: $\max(l - r, 0)$
- **INTERSECT ALL** (very obscure): $\min(l, r)$

```sql
select n_name from nation where n_regionkey = 2
union all
select n_name from nation where n_regionkey in (1, 2)
intersect all
select n_name from nation where n_regionkey < 3
except all
select n_name from nation where n_nationkey = 21;
```
Miscellaneous Useful Constructs

- **case** (conditional expressions)

  ```sql
  select case when n_nationkey > 5
    then 'large'
    else 'small'
  end
  from nation;
  ```

- **coalesce**(a, b): replace NULL with some other value

- **cast** (explicit type conversion)

- **generate_series**(begin, end)

- **random** (random float from 0 to 1):

  ```sql
  select cast(random() * 6 as integer) + 1
  from generate_series(1, 10);
  ```

  -- 10 dice rolls
Working With Strings

- **concatenation:**
  ```sql
  select 'a' || 'b';
  ```

- **simple string matching:**
  ```sql
  select 'abcfoo' like 'abc%';
  ```

- **regexp string matching:**
  ```sql
  select 'abcabc' ~ '(abc)*';
  ```

- **extract substring:**
  ```sql
  select substring('abcfoo' from 3 for 2);
  ```

- **regexp-based replacement:** (str, pattern, replacement, flags)
  ```sql
  select regexp_replace('ababfooab', '(ab)+', 'xy', 'g');
  ```
Sampling

- sampling modes: bernoulli (pick random tuples) or system (pick random pages)
- set random seed with optional repeatable setting
- supported by PostgreSQL $\geq 9.5$:

```sql
select *
from nation tablesample bernoulli(5) -- 5 percent
    repeatable (9999);
```

- it is also possible to get arbitrary tuples:

```sql
select *
from nation
limit 10; -- 10 arbitrary rows
```
compute the average of totalprice using a sample of 1% of all orders
Views and Common Table Expressions

- like functions in “normal” programming languages
- code reuse, abstraction, readability
- in PostgreSQL, WITH is an optimization breaker, views are not

```sql
create view nation_europe as
    select nation.*
    from nation, region
    where n_regionkey = r_regionkey
    and r_name = 'EUROPE';

with old_orders as (  
    select *  
    from orders  
    where o_orderdate < date '2000-01-01')
select count(*)
from nation_europe, customer, old_orders
where n_nationkey = c_nationkey
and c_custkey = o_custkey;
```
Recursive Common Table Expressions

- called recursive but is actually iteration
- traverse hierarchical data of arbitrary depth (joins only allow a constant number of steps)

```sql
WITH recursive r (i) AS (
    SELECT 1 -- non-recursive term
    UNION ALL
    SELECT i+1 FROM r WHERE i < 5) -- recursive term
SELECT * FROM r;
```

- algorithm:
  ```sql
  workingTable = evaluateNonRecursive()
  output workingTable
  while workingTable is not empty
      workingTable = evaluateRecursive(workingTable)
  output workingTable
  ```
WITH RECURSIVE ... UNION ALL

with recursive
animals (id, name, parent) as (values (1, 'animal', null),
(2, 'mammal', 1), (3, 'giraffe', 2), (4, 'tiger', 2),
(5, 'reptile', 1), (6, 'snake', 5), (7, 'turtle', 5),
(8, 'green sea turtle', 7)),

r as (select * from animals where name = 'turtle'
union all
select animals.*
from r, animals
where animals.id = r.parent)

select * from r;
• compute all descendants of 'reptile'
• compute 10! using recursion
• compute the first 20 Fibonacci numbers \( (F_1 = 1, \ F_2 = 1, \ F_n = F_{n-1} + F_{n-2}) \)
Recursive Common Table Expressions with UNION

- for graph-like data UNION ALL may not terminate
- with recursive [non-recursive] union [recursive]
- allows one to traverse cyclic structures
- algorithm:
  \[
  \text{workingTable} = \text{unique(evaluateNonRecursive())} \\
  \text{result} = \text{workingTable} \\
  \text{while workingTable is not empty} \\
  \hspace{1em} \text{workingTable} = \text{unique(evaluateRecursive(workingTable)))} \setminus \text{result} \\
  \hspace{1em} \text{result} = \text{result} \cup \text{workingTable} \\
  \text{output result}
  \]
WITH RECURSIVE • UNION

with recursive
friends (a, b) as (values ('Alice', 'Bob'), ('Alice', 'Carol'),
('Carol', 'Grace'), ('Carol', 'Chuck'), ('Chuck', 'Grace'),
('Chuck', 'Anne'), ('Bob', 'Dan'), ('Dan', 'Anne'), ('Eve', 'Adam')), friendship (name, friend) as -- friendship is symmetric
(select a, b from friends union all select b, a from friends),
r as (select 'Alice' as name
union
select friendship.name from r, friendship
where r.name = friendship.friend)
select * from r;
Window Functions

- very versatile feature: time series analysis, ranking, top-k, percentiles, moving averages, cumulative sums
- window functions are evaluated after most other clauses (including group by) but before order by
- in contrast to aggregation, window functions do not change the input, they only compute additional columns
Window Functions: Concepts

```sql
select o_custkey, o_orderdate,
       sum(o_totalprice) over -- window function
       (partition by o_custkey -- partitioning clause
        order by o_orderdate -- ordering clause
        range between unbounded preceding
               and current row) -- framing clause
from customer;
```

```
order by
partition by
```

```
frame
```

```
```

```
```

```
Window Functions Framing That Ignore Framing

- **ranking:**
  - `rank()`: rank of the current row with gaps
  - `dense_rank()`: rank of the current row without gaps
  - `row_number()`: row number of the current row
  - `ntile(n)`: distribute evenly over buckets (returns integer from 1 to n)

- **distribution:**
  - `percent_rank()`: relative rank of the current row \((\frac{\text{rank} - 1}{\text{total rows} - 1})\)
  - `cume_dist()`: relative rank of peer group \((\frac{\text{number of rows preceding or peer with current row}}{\text{total rows}})\)

- **navigation in partition:**
  - `lead(expr, offset, default)`: evaluate `expr` on preceding row in partition
  - `lag(expr, offset, default)`: evaluate `expr` on following row in partition
- determine medals based on number of orders, example output:

<table>
<thead>
<tr>
<th>custkey</th>
<th>count</th>
<th>medal</th>
</tr>
</thead>
<tbody>
<tr>
<td>8761</td>
<td>36</td>
<td>gold</td>
</tr>
<tr>
<td>11998</td>
<td>36</td>
<td>gold</td>
</tr>
<tr>
<td>8362</td>
<td>35</td>
<td>bronze</td>
</tr>
<tr>
<td>4339</td>
<td>35</td>
<td>bronze</td>
</tr>
<tr>
<td>388</td>
<td>35</td>
<td>bronze</td>
</tr>
<tr>
<td>3151</td>
<td>35</td>
<td>bronze</td>
</tr>
<tr>
<td>9454</td>
<td>35</td>
<td>bronze</td>
</tr>
</tbody>
</table>
Window Functions: Framing

- current row: the current row (including all peers in range mode)
- unbounded preceding: first row in the partition
- unbounded following: last row in the partition

(rows) between 3 preceding and 3 following

range between 3 preceding and 3 following

- default frame (when an order by clause was specified): range between unbounded preceding and current row
- default frame (when no order by clause was specified): range between unbounded preceding and unbounded following
- complex frame specifications are not yet supported by PostgreSQL
Window Functions With Framing

- aggregates \((\text{min}, \text{max}, \text{sum}, \ldots)\): compute aggregate over all tuples in current frame
- navigation in frame:
  first_value(expr), last_value(expr), nth_value(expr, nth):
evaluate expr on first/last/nth row of the frame
- compute the cumulative customer spending \( \text{sum}(o\text{.totalprice}) \) over time \( o\text{.orderdate} \)

- for each customer from GERMANY compute the cumulative spending \( \text{sum}(o\text{.totalprice}) \) by year \( \text{extract(year from o\text{.orderdate})} \), example output:

```
custkey | yr  | running_sum
--------+-----+---------------
  62    | 1992| 169991.32
  62    | 1993| 344376.79
  62    | 1994| 433638.98
  62    | 1995| 960047.31
  62    | 1996| 1372061.28
  62    | 1997| 1658247.25
  62    | 1998| 2055669.94
  71    | 1992| 403017.41
  71    | 1993| 751256.86
  71    | 1994| 1021446.72
  71    | 1995| 1261012.10
```
Statistical Aggregates

- `stddev_samp(expr)`: standard deviation
- `corr(x, y)`: correlation
- `regr_slope(y, x)`: linear regression slope
- `regr_intercept(y, x)`: linear regression intercept
Ordered-Set Aggregates

- **mode()**: most frequently occurring value
- **percentile_disc(p)**: compute discrete percentile \((p \in [0, 1])\)
- **percentile_cont(p)**: compute continuous percentile \((p \in [0, 1])\), may interpolate, only works on numeric data types
- aggregate functions with a special syntax that require sorting

```sql
select percentile_cont(0.5)
  within group (order by o_totalprice)
from orders;
```

```sql
select o_custkey,
  percentile_cont(0.5) within group (order by o_totalprice)
from orders
group by o_custkey;
```
Grouping Sets, Rollup, Cube

- aggregate across multiple dimensions, e.g., revenue by year, by customer, by supplier
- specify multiple groupings:
  group by grouping sets ((a, b), (a), ())
- hierarchical groupings:
  group by rollup (a, b)
- both are equivalent to:

  ```sql
  select a, b, sum(x) from r group by a, b
  union all
  select a, null, sum(x) from r group by a
  union all
  select null, null, sum(x) from r;
  ```

- all \(2^n\) groupings:
  group by cube (a, b) is equivalent to
  group by grouping sets ((a, b), (a), (b), ())
- aggregate revenue ($\text{sum(o\_totalprice)}$): total, by region ($r\_name$), by name ($n\_name$), example output:

```
<table>
<thead>
<tr>
<th>revenue</th>
<th>region</th>
<th>nation</th>
</tr>
</thead>
<tbody>
<tr>
<td>836330704.31</td>
<td>AFRICA</td>
<td>ALGERIA</td>
</tr>
<tr>
<td>902849428.98</td>
<td>AFRICA</td>
<td>ETHIOPIA</td>
</tr>
<tr>
<td>784205751.27</td>
<td>AFRICA</td>
<td>KENYA</td>
</tr>
<tr>
<td>893122668.52</td>
<td>AFRICA</td>
<td>MOROCCO</td>
</tr>
<tr>
<td>852278134.31</td>
<td>AFRICA</td>
<td>MOZAMBIQUE</td>
</tr>
<tr>
<td>4268786687.39</td>
<td>AFRICA</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>21356596030.63</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
References

- SQL reference (PostgreSQL): https://www.postgresql.org/docs/current/static/sql.html
- Joe Celko’s SQL for Smarties, Joe Celko, 5th edition, 2014
- SQL cookbook, Anthony Molinaro, 2005
Why Database Clusters?

Limitations of a single machine:

- space (a few terabytes)
- performance (hundreds of GFLOPS)
- geo-locality (speed of light)
- no redundancy (in case of failures)

Therefore, we need to use multiple machines.
Properties of Database Clusters

- transparent for the user (cluster presents itself as a single system)
- consistency vs. fussy consistency (ACID vs. BASE)
Properties of Database Clusters

ACID

- **Atomicity**: a transaction is either executed completely or not at all
- **Consistency**: transactions can not introduce inconsistencies
- **Isolation**: concurrent transactions can not see each other
- **Durability**: all changes of a transaction are durable
Properties of Database Clusters

BASE

- **Basically Available**: requests may fail
- **Soft state**: the system state may change over time
- **Eventual consistency**: once the system stops receiving input, it will eventually become consistent
Properties of Database Clusters

Providing ACID guarantees slows down the database, as more synchronization is required. **BUT:**

- BASE does not provide a consistent view of the data, which can lead to subtle bugs or confuse customers
- BASE can simply not be used for many critical systems (e.g., banking)
- before using BASE, be sure that you really do need the performance and don’t need ACID guarantees. otherwise, you will end up implementing ACID on the application layer
End of Transaction Handling

Just like in a non-distributed system, a transaction has to be atomic. There are two options for ending a transaction:

- **commit**: the transaction was successful and all constraints are fulfilled: it has to become persistent and visible to all nodes of the cluster
- **abort**: the transaction failed or violates a constraint: it has to be undone on all nodes of the cluster

Problem: the nodes of a distributed system can crash independently
**End of Transaction Handling**

Solution: Two Phase Commit Protocol (2PL).

- 2PL ensures the atomic property of distributed transactions
- one node acts as a coordinator
- it allows $n$ agents of a system $A_1, A_2, .., A_n$, which participated in a transaction $T$ to either all persist the changes of $T$ or all discard the changes of $T$
Two Phase Commit

Message flow in 2PL with 4 agents:
Two Phase Commit

Potential points of failure in 2PL:

- crash of the coordinator
- crash of an agent
- lost messages
Replication

Every machine in the cluster has a (consistent) copy of the entire dataset.

- improved query throughput
- can accommodate node failures
- solves locality issue
Horizontal Partitioning (Sharding)

Every machine in the cluster has a chunk of the dataset.

- improved query runtimes (especially if no cross-shard communication is required)
- performance heavily relies on the used interconnect
- can accommodate datasets that exceed the capacity of a machine
Horizontal Partitioning (Sharding) - Example

```
select Title
from Vorlesungen v, Professoren p
where v.gelesenVon = p.persNr
  and p.rang = 'C4';
```
Horizontal Partitioning (Sharding) - Example (Optimized)

```sql
select Title
from Vorlesungen v, Professoren p
where v.gelesenVon = p.persNr
  and p.rang = 'C4';
```
Shard Allocation

Shards can be allocated to the cluster machines in different ways.

- every machine receives one shard
- every machine receives multiple shards (e.g., MemSQL allocates 8 shards to a leaf node)
- having more shards than machines results in better skew handling
Replication and Sharding

Replication and sharding can be applied together.

- benefits of both are combined
- however, this leads to an increased resource consumption
Joins

Joins in a replicated setting are easy (i.e., not different from local joins). What is more challenging are joins when data is sharded across multiple machines. Join types include:

- co-located joins
- distributed join (involves cross-shard communication)
- broadcast join (involves cross-shard communication)
Co-Located Joins

One option is to shard the tables that need to be joined on the respective join column. That way, joins can be computed locally and the individual machines do not need to communicate with each other. Eventually, the individual results are send to a master node that combines the results.
Distributed Join

Re-distribute both tables based on the corresponding join columns to eventually compute the join locally.
Broadcast Join

Send smaller relation to all machines to compute the join locally.
Counting the Sum

Let's consider summing up the quantity in `lineitem`, again. Assume `lineitem` does not fit into the DRAM of a single machine. Depending on the used interconnect, it turns out that reading `lineitem` from remote machines is faster than reading it from a local SSD.

<table>
<thead>
<tr>
<th>Interconnect</th>
<th>Bandwidth</th>
<th>Query Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>FDR InfiniBand</td>
<td>7GB/s</td>
<td>0.1s</td>
</tr>
<tr>
<td>10GB ethernet</td>
<td>1.25GB/s</td>
<td>0.6s</td>
</tr>
<tr>
<td>1GB ethernet</td>
<td>125MB/s</td>
<td>6s</td>
</tr>
<tr>
<td>Rotating disk</td>
<td>200MB/s</td>
<td>3.6s</td>
</tr>
<tr>
<td>SATA SSD</td>
<td>500MB/s</td>
<td>1.6s</td>
</tr>
<tr>
<td>PCIe SSD</td>
<td>2GB/s</td>
<td>0.36s</td>
</tr>
<tr>
<td>DRAM</td>
<td>20GB/s</td>
<td>0.04s</td>
</tr>
</tbody>
</table>
Counting the Sum - Sharding

Lower query latencies by sharding the data to multiple nodes.
- each node only stores a partition of lineitem
- every node sums up its partition of lineitem
- a master node sums up the local sums
Vanilla PostgreSQL

What can we do with vanilla PostgreSQL?

- PostgreSQL supports read replicas
  - from version 9.6, these replicas are kept in sync, allowing for consistent reads
  - allows for load-balancing read workloads and thus improved query throughput
- PostgreSQL neither supports data distribution nor distributed queries
Replication in PostgreSQL

- **Master**
- **Slave 1**
- **Slave 2**
- **Application 1**
- **Application 2**
- **Application 3**

Transactions flow from Master to Slave 1 and Slave 2. Queries flow from Application 1, Application 2, and Application 3 to Slave 2. The Log is replicated across all nodes.
Greenplum to the rescue!

Pivotal Greenplum is a distributed open-source database based on PostgreSQL.

- supports data distribution and partitioning
- master and segment (child) nodes
  - master nodes contain the schemas
  - segment nodes contain the data
- additional standby master and secondary (mirror) segment nodes
Greenplum: Distribution/Partitioning Schemes

Distribution/partitioning schemes can be specified when creating a table.

- **DISTRIBUTED BY** specifies how data is spread (sharded) across segments
  - random (round-robin) or hash-based distribution
- **PARTITION BY** specifies how data is partitioned within a segment
  - list of values or range (numeric or date)
  - subpartitioning
Greenplum: Distribution Example

A table with a hash-based distribution:

CREATE TABLE products
(name varchar(40),
prod_id integer,
supplier_id integer)
DISTRIBUTED BY (prod_id);

A table with a random (round-robin) distribution:

CREATE TABLE random_stuff
(things text,
doodads text,
etc text)
DISTRIBUTED RANDOMLY;

Greenplum: Partitioning Example

URL: http://gpdb.docs.pivotal.io/4350/admin_guide/ddl/ddl-partition.html
MemSQL

MemSQL is yet another (distributed) database system.

- compatible with MySQL
- data can be sharded across multiple machines
- in addition to sharding, tables can be replicated to all machines (reference tables) allowing for broadcast joins
- also supports distributed and co-located joins
A MemSQL cluster consists of aggregator and leaf nodes.

- Aggregator nodes handle metadata, distribute queries, and aggregate results.
- Aggregator nodes are also responsible for cluster monitoring and failover.
- Leaf nodes act as the storage layer of the cluster and execute SQL queries.
By default, MemSQL shards tables by their primary key. A manual shard key can be specified as follows:

```sql
CREATE TABLE clicks (  
  click_id BIGINT AUTO_INCREMENT,  
  user_id INT,  
  page_id INT,  
  ts TIMESTAMP,  
  SHARD KEY (user_id),  
  PRIMARY KEY (click_id, user_id)  
);  
```

[3]https://docs.memsql.com/docs/distributed-sql
Conclusion

- A distributed system provides access to a bunch of machines and makes them look like a single one to the user.
- These systems naturally have more resources (e.g., compute power, storage) than a single machine.
- This makes clusters attractive for developers to simply put any data there.
- **But**: the synchronization of these resources may be expensive.