CS5412: LECTURE 18
THE SQL QUERY LANGUAGE

Ken Birman

Fall, 2022

SQL INTRODUCTION

Standard language for querying and manipulating data

Structured **Q**uery **L**anguage

Many of today's slides were shared by the instructors of CSE544 at U. Washington

BASIC CONCEPTS

A **relational database** is a set of tables (“**relations**”) with a layout (“**schema**”) that, in modern cloud settings, could be huge – many columns (“**attributes**”)

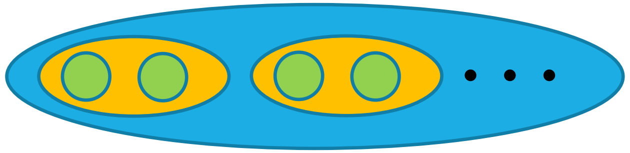
Each relation holds rows (“**tuples**”). In a big-data setting, there could be billions of rows and thousands of attributes. Some fields might hold nulls.

Conceptually, a database is not sharded – the “model” is expressed as if there was a single and complete database shared and seen by all users.

RELATIONAL DATABASE? OR KEY-VALUE STORE?

On a slide they look kind of similar….

Relational Databases



Key-value store

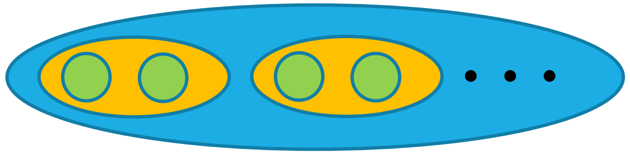
BUT SLIDES DON'T CONVEY SCALE WELL

The key-value store could be running on 2500 servers, split into 1250 shards that are each holding a huge amount of data in memory.

The database system would probably run on just a few servers, maybe 3 to 5 per database (that illustration showed a few databases playing distinct roles). They could be interacting with vast amounts of storage, far more data than can be held in memory even at "key-value scale", but the database itself probably doesn't run on a huge number of servers.

RELATIONAL DATABASE? OR KEY-VALUE STORE?

Relational Databases



Key-value store

A relational database system has a set of very sophisticated servers that host structured data ("relations"), plan and execute SQL queries.

- Provides what are called the ACID guarantees (A for atomicity).
- Often use locking for concurrency control
- Runs a two-phase commit protocol at the end of any updates

A key-value store is simpler, only offers **put/get/watch** with O(1) performance. No locking or transactions, except perhaps "one-shot" atomic actions

A VERY BRIEF HISTORY OF THE AREA

Databases were the dominant form of data management and computing from the 1980's until around 2000.

But the cloud took "big data" to a totally different scale, larger by 10,000x and at the same time, with a very different pattern of work

This led to emergence of key-value stores and their NoSQL model

GENUINE DATABASES HAVE TWO HUGE ADVANTAGES: SIMPLE MODEL, ACID GUARANTEES

Very natural to think in terms of tables (**relations**), transformations from table to table. SQL has simple ways to express connections between tabular data sets and to perform sophisticated data transformations.

Users (mostly) don't worry about efficiency. The server executes requests efficiently, figures out which fast index structures to build, etc.

- *But sometimes, a little help from a well-informed user pays off in big ways*
- *No complex technology ever is completely foolproof and self-managed*

GENUINE DATABASES HAVE TWO HUGE ADVANTAGES: SIMPLE MODEL, ACID GUARANTEES

SQL queries are executed "as if" the query was running in an idle system.

ACID stands for
atomic, **c**onsistent, **i**solated, **d**urable.

ACID properties: SQL operations are **atomic** (either executed completely, or any partial updates roll back), **durable** (database won't forget things) and the database server is able to stay busy (high level of **concurrency** while maintain these properties).

MONOLITHIC DATABASES RUNNING ON PARALLEL SERVERS DON'T HANDLE BIG DATA VERY WELL

We learned about Jim Gray's study early in the course. He looked at one big database somehow replicated across N servers. This was standard in the 1990's.

He explained that a drastic slowdown occurs: Overheads rise as $O(N^3 T^5)$

The issue is that with an uncontrolled mix of transactions, locking conflicts (which sometimes trigger aborts/rollback) force the database to work harder and harder to do the same tasks. Leads to *sharding*

REMINDER: SHARDING A DATABASE

Splits the big database into multiple *independent* databases. Queries can run on any one database, but never span across a set of them

In this sharded model, we do get scalability. But we've lost the ability to think of our data as if it lived in one big pool.

Today, key-value sharding is mostly used with DHTs. Databases are still mostly monolithic, not sharded, but we use them very carefully!

… YET DATABASES AREN'T GONE!!!

Relational Databases

We do need to shield them from excessive load, to avoid collapse. Often we filter all the reads and only send them the updates.

And we host them deep in the cloud, with layers of functions and μ-services to absorb as much work as possible

But at the end of the pipeline, you still find massive enterprise databases in any major system. They continue to be one of the most important cloud components, even if key-value DHTs handle large categories of work!

SQL LANGUAGE

Used to access or update relational (tabular data)

In modern settings, the tables can be huge. Database will automatically fragment the data and parallelize the queries and updates for speed

You can think of the database as a compiler: it translates your SQL code into a plan, then executes that plan for you. Like Python, but the data you care about lives "in" the database, and the program runs "on" it.

EXAMPLE: A RELATION

Relation name

Attribute name

Product

PName	Price	Category	Manufacturer
Gizmo	$19.99	Gadgets	GizmoWorks
Powergizmo	$29.99	Gadgets	GizmoWorks
SingleTouch	$149.99	Photography	Canon
MultiTouch	$203.99	Household	Hitachi

Tuple

NULLS IN SQL

Whenever we don’t have a value, we can put a NULL

Can mean many things:

- Value does not exists
- Value exists but is unknown
- Value not applicable
- Etc.

The schema specifies for each attribute if can be null (nullable attribute) or not

How does SQL cope with tables that have NULLs ?

NULL VALUES

If x= NULL then 4*(3-x)/7 is still NULL

If x= NULL then x="Joe"   is UNKNOWN

In SQL there are three truth values:

- FALSE = 0
- UNKNOWN
- TRUE = 1

SQL QUERY

Basic form: (plus many many more bells and whistles)

```
SELECT  <attributes>
FROM    <one or more relations>
WHERE   <conditions>
```

SIMPLE SQL QUERY

Product

PName	Price	Category	Manufacturer
Gizmo	$19.99	Gadgets	GizmoWorks
Powergizmo	$29.99	Gadgets	GizmoWorks
SingleTouch	$149.99	Photography	Canon
MultiTouch	$203.99	Household	Hitachi

```
SELECT   *
FROM     Product
WHERE    category='Gadgets'
```

PName	Price	Category	Manufacturer
Gizmo	$19.99	Gadgets	GizmoWorks
Powergizmo	$29.99	Gadgets	GizmoWorks

"selection"

SIMPLE SQL QUERY

Product

PName	Price	Category	Manufacturer
Gizmo	$19.99	Gadgets	GizmoWorks
Powergizmo	$29.99	Gadgets	GizmoWorks
SingleTouch	$149.99	Photography	Canon
MultiTouch	$203.99	Household	Hitachi

```
SELECT   PName, Price, Manufacturer
FROM     Product
WHERE    Price > 100
```

PName	Price	Manufacturer
SingleTouch	$149.99	Canon
MultiTouch	$203.99	Hitachi

“selection” and “projection”

NOTATION

Input Schema. Here the designer indicates that Pname should be used as a unique key for each tuple

Product(<u>PName</u>, Price, Category, Manfacturer)

```
SELECT   PName, Price, Manufacturer
FROM     Product
WHERE    Price > 100
```

Answer(<u>PName</u>, Price, Manfacturer)

Output Schema. We could have named the output but here it was left anonymous

MANY OPTIONS FOR WHERE THE ANSWER LIVES

You can just type the query, and it will print the answer

You can tell the database to save the result as a new relation, with a new name. You just write newname = query.

You can ask the database to remember the query and recompute the result as needed. This is called a *dynamically materialized view…* Like a virtual relation that is auto-updated when underlying data changes.

DISTINCT: A KEYWORD USED FOR ELIMINATING DUPLICATES

```
SELECT   DISTINCT category
FROM     Product
```

⇨

Category
Gadgets
Photography
Household

Compare to:

```
SELECT   category
FROM     Product
```

⇨

Category
Gadgets
Gadgets
Photography
Household

ORDERING THE RESULTS

```
SELECT   pname, price, manufacturer
FROM     Product
WHERE    category='gizmo' AND price > 50
ORDER BY price, pname
```

Ties are broken by the second attribute on the ORDER BY list, etc.

Ordering is ascending, unless you specify the DESC keyword.

PName	Price	Category	Manufacturer
Gizmo	$19.99	Gadgets	GizmoWorks
Powergizmo	$29.99	Gadgets	GizmoWorks
SingleTouch	$149.99	Photography	Canon
MultiTouch	$203.99	Household	Hitachi

```
SELECT   DISTINCT Category
FROM     Product
ORDER BY Category
```

⇨ ?

```
SELECT   Category
FROM     Product
ORDER BY PName
```

⇨ ?

```
SELECT   DISTINCT Category
FROM     Product
ORDER BY PName
```

⇨ ?

PName	Price	Category	Manufacturer
Gizmo	$19.99	Gadgets	GizmoWorks
Powergizmo	$29.99	Gadgets	GizmoWorks
SingleTouch	$149.99	Photography	Canon
MultiTouch	$203.99	Household	Hitachi

```
SELECT   DISTINCT Category
FROM      Product
ORDER BY Category
```

Category
Gadgets
Household
Photography

```
SELECT   Category
FROM      Product
ORDER BY  PName
```

Category
Gadgets
Household
Gadgets
Photography

```
SELECT   DISTINCT Category
FROM      Product
ORDER BY PName
```

Category
Gadgets
Household
Photography

KEYS AND FOREIGN KEYS

Company

Key

CName	StockPrice	Country
GizmoWorks	25	USA
Canon	65	Japan
Hitachi	15	Japan

Product

PName	Price	Category	Manufacturer
Gizmo	$19.99	Gadgets	GizmoWorks
Powergizmo	$29.99	Gadgets	GizmoWorks
SingleTouch	$149.99	Photography	Canon
MultiTouch	$203.99	Household	Hitachi

Foreign key: A key from CName in Company shows up in Product with a different name

JOINS ARISE WHEN WE WRITE QUERIES THAT OPERATE ON TWO OR MORE RELATIONS

Product (<u>pname</u>, price, category, manufacturer)

Company (<u>cname</u>, stockPrice, country)

Find all products under $200 manufactured in Japan; return their names and prices.

Join between Product and Company

```
SELECT   PName, Price
FROM     Product, Company
WHERE    Manufacturer=CName AND Country='Japan'
           AND Price < 200
```

JOINS

Product

PName	Price	Category	Manufacturer
Gizmo	$19.99	Gadgets	GizmoWorks
Powergizmo	$29.99	Gadgets	GizmoWorks
SingleTouch	$149.99	Photography	Canon
MultiTouch	$203.99	Household	Hitachi

Company

Cname	StockPrice	Country
GizmoWorks	25	USA
Canon	65	Japan
Hitachi	15	Japan

```
SELECT   PName, Price
FROM     Product, Company
WHERE    Manufacturer=CName AND Country='Japan'
           AND Price < 200
```

PName	Price
SingleTouch	$149.99

NULL VALUES

C1 AND C2 = min(C1, C2)

C1 OR C2 = max(C1, C2)

NOT C1 = 1 – C1

```
SELECT *
FROM Person
WHERE  (age < 25) AND
           (height > 6 OR weight > 190)
```

E.g.
age=20
heigth=NULL
weight=200

Rule in SQL: include only tuples that yield TRUE

OUTER JOINS

Explicit joins in SQL = "inner joins":

Product(name, category)

Purchase(prodName, store)

```
SELECT Product.name, Purchase.store
FROM   Product JOIN Purchase ON
                 Product.name = Purchase.prodName
```

Same as:

```
SELECT Product.name, Purchase.store
FROM   Product, Purchase
WHERE  Product.name = Purchase.prodName
```

But Products that never sold will be lost !

OUTER JOINS

Left outer joins in SQL:

Product(name, category)

Purchase(prodName, store)

```
SELECT Product.name, Purchase.store
FROM      Product LEFT OUTER JOIN Purchase ON
                    Product.name = Purchase.prodName
```

Product

Name	Category
Gizmo	gadget
Camera	Photo
OneClick	Photo

Purchase

ProdName	Store
Gizmo	Wiz
Camera	Ritz
Camera	Wiz

Name	Store
Gizmo	Wiz
Camera	Ritz
Camera	Wiz
OneClick	NULL

APPLICATION

Compute, for each product, the total number of sales in 'September'

Product(name, category)

Purchase(prodName, month, store)

```
SELECT Product.name, count(*)
FROM     Product, Purchase
WHERE   Product.name = Purchase.prodName
        and  Purchase.month = 'September'
GROUP BY Product.name
```

What's wrong ?   … no sales? Not listed!

APPLICATION

Compute, for each product, the total number of sales in 'September'

Product(name, category)

Purchase(prodName, month, store)

```
SELECT Product.name, count(*)
FROM      Product LEFT OUTER JOIN Purchase ON
                    Product.name = Purchase.prodName
               and  Purchase.month = 'September'
GROUP BY Product.name
```

Now we also get the products sold in 0 quantity

OUTER JOINS

Left outer join:

- Include the left tuple even if there's no match

Right outer join:

- Include the right tuple even if there's no match

Full outer join:

- Include the both left and right tuples even if there's no match

A SUBTLETY ABOUT JOINS

Product

Name	Price	Category	Manufacturer
Gizmo	$19.99	Gadgets	GizmoWorks
Powergizmo	$29.99	Gadgets	GizmoWorks
SingleTouch	$149.99	Photography	Canon
MultiTouch	$203.99	Household	Hitachi

Company

Cname	StockPrice	Country
GizmoWorks	25	USA
Canon	65	Japan
Hitachi	15	Japan

```
SELECT   Country
FROM      Product, Company
WHERE   Manufacturer=CName AND Category='Gadgets'
```

What is
the problem ?
What's the
solution ?

Country
??
??

A SUBTLETY ABOUT JOINS

Product

Name	Price	Category	Manufacturer
Gizmo	$19.99	Gadgets	GizmoWorks
Powergizmo	$29.99	Gadgets	GizmoWorks
SingleTouch	$149.99	Photography	Canon
MultiTouch	$203.99	Household	Hitachi

Company

Cname	StockPrice	Country
GizmoWorks	25	USA
Canon	65	Japan
Hitachi	15	Japan

```
SELECT   Country
FROM     Product, Company
WHERE    Manufacturer=CName AND Category='Gadgets'
```

Country
USA
USA
Japan
Japan

DISTINCT would have
given one instance per country

AMBIGUOUS ATTRIBUTE NAMES

Person(<u>pname</u>, address, worksfor)

Company(<u>cname</u>, address)

```
SELECT   DISTINCT pname, address
FROM      Person, Company
WHERE   worksfor = cname
```

SQL will complain: which address attribute is being referenced ?

```
SELECT   DISTINCT Person.pname, Company.address
FROM      Person, Company
WHERE   Person.worksfor = Company.cname
```

Better, but "wordy"

```
SELECT   DISTINCT x.pname, y.address
FROM      Person AS x, Company AS y
WHERE   x.worksfor = y.cname
```

Code is more "concise"

CORRELATED QUERY

Product (pname, price, category, maker, year)

Find products (and their manufacturers) that are more expensive than all products made by the same manufacturer before 1972

```
SELECT DISTINCT  pname, maker
FROM      Product AS x
WHERE  price > ALL  (SELECT  price
                     FROM     Product AS y
                     WHERE  x.maker = y.maker AND y.year < 1972);
```

Very powerful ! But in this case, using an “aggregator” would have been simpler and faster

AGGREGATION

```
SELECT  avg(price)
FROM    Product
WHERE   maker="Toyota"
```

```
SELECT  count(*)
FROM    Product
WHERE   year > 1995
```

SQL supports several aggregation operations:

sum, count, min, max, avg

AGGREGATION: COUNT

COUNT applies to duplicates, unless otherwise stated:

```
SELECT  Count(category)
FROM    Product
WHERE   year > 1995
```

same as Count(*)

We probably want:

```
SELECT  Count(DISTINCT category)
FROM    Product
WHERE   year > 1995
```

SIMPLE AGGREGATIONS

Purchase

Product	Date	Price	Quantity
Bagel	10/21	1	20
Banana	10/3	0.5	10
Banana	10/10	1	10
Bagel	10/25	1.50	20

```
SELECT  Sum(price * quantity)
FROM    Purchase
WHERE   product = 'bagel'
```

⇨

50 (= 20+30)

GROUPING AND AGGREGATION

Purchase(product, date, price, quantity)

Find total sales after 10/1/2005 per product.

```
SELECT      product, Sum(price*quantity) AS TotalSales
FROM        Purchase
WHERE       date > '10/1/2005'
GROUP BY    product
```

GROUPING AND AGGREGATION

1. Compute the FROM and WHERE clauses.

2. Group by the attributes in the GROUPBY

3. Compute the SELECT clause: grouped attributes and aggregates.

1&2. FROM-WHERE-GROUPBY

Product	Date	Price	Quantity
Bagel	10/21	1	20
Bagel	10/25	1.50	20
Banana	10/3	0.5	10
Banana	10/10	1	10

3. SELECT

Product	Date	Price	Quantity
Bagel	10/21	1	20
Bagel	10/25	1.50	20
Banana	10/3	0.5	10
Banana	10/10	1	10

Product	TotalSales
Bagel	50
Banana	15

```
SELECT     product, Sum(price*quantity) AS TotalSales
FROM        Purchase
WHERE      date > '10/1/2005'
GROUP BY  product
```

ANOTHER EXAMPLE

What does it mean ?

```
SELECT    product,
            sum(price * quantity) AS SumSales
            max(quantity) AS MaxQuantity
FROM       Purchase
GROUP BY product
```

HAVING CLAUSE

Same query, except that we consider only products that had at least 100 buyers.

```
SELECT      product, Sum(price * quantity)
FROM         Purchase
WHERE       date > '10/1/2005'
GROUP BY product
HAVING      Sum(quantity) > 30
```

HAVING clause contains conditions on aggregates.

GENERAL FORM OF GROUPING AND AGGREGATION

```
SELECT    S
FROM      R1,…,Rn
WHERE     C1
GROUP BY  a1,…,ak
HAVING    C2
```

S = may contain attributes a1,…,ak and/or any aggregates but NO OTHER ATTRIBUTES

C1 = is any condition on the attributes in R1,…,Rn

C2 = is any condition on aggregate expressions

GENERAL FORM OF GROUPING AND AGGREGATION

```
SELECT    S
FROM      R1,…,Rn
WHERE     C1
GROUP BY  a1,…,ak
HAVING    C2
```

Evaluation steps:

1. Evaluate FROM-WHERE, apply condition C1
2. Group by the attributes $a_1,\ldots,a_k$
3. Apply condition C2 to each group (may have aggregates)
4. Compute aggregates in S and return the result

2. QUANTIFIERS

1. Find *the other* companies: i.e. s.t. <u>some</u> product ≥ 100

```
SELECT DISTINCT  Company.cname
FROM      Company
WHERE  Company.cname IN (SELECT Product.company
                          FROM Product
                          WHERE Product.price >= 100
```

2. Find all companies s.t. <u>all</u> their products have price < 100

```
SELECT DISTINCT  Company.cname
FROM      Company
WHERE  Company.cname NOT IN (SELECT Product.company
                              FROM Product
                              WHERE Product.price >= 100
```

3. GROUP-BY V.S. NESTED QUERY

Find authors who wrote 10 documents:

Attempt 1: with nested queries

Author(<u>login</u>,name)

Wrote(login,url)

```
SELECT DISTINCT Author.name
FROM        Author
WHERE       count(SELECT Wrote.url
                  FROM Wrote
                  WHERE Author.login=Wrote.login)
               > 10
```

This is SQL by a novice

3. GROUP-BY V.S. NESTED QUERY

Find all authors who wrote at least 10 documents:

Attempt 2: SQL style (with GROUP BY)

```
SELECT     Author.name
FROM       Author, Wrote
WHERE      Author.login=Wrote.login
GROUP BY   Author.name
HAVING     count(wrote.url) > 10
```

This is SQL by an expert

No need for DISTINCT: automatically from GROUP BY

3. GROUP-BY V.S. NESTED QUERY

Author(login,name)

Wrote(login,url)

Mentions(url,word)

Find authors with vocabulary ≥ 10000 words:

```
SELECT     Author.name
FROM       Author, Wrote, Mentions
WHERE      Author.login=Wrote.login AND Wrote.url=Mentions.url
GROUP BY   Author.name
HAVING     count(distinct Mentions.word) > 10000
```

MODIFYING THE DATABASE

Three kinds of modifications

Insertions

Deletions

Updates

Sometimes they are all called “updates”

INSERTIONS

General form:

```
INSERT   INTO   R(A1,…., An)   VALUES  (v1,…., vn)
```

Example: Insert a new purchase to the database:

```
INSERT  INTO  Purchase(buyer, seller, product, store)
        VALUES  ('Joe', 'Fred', 'wakeup-clock-espresso-machine',
                 'The Sharper Image')
```

Missing attribute → NULL.
May drop attribute names if give them in order.

INSERTIONS

```
INSERT   INTO   PRODUCT(name)

     SELECT  DISTINCT  Purchase.product
     FROM        Purchase
     WHERE   Purchase.date > “10/26/01”
```

The query replaces the VALUES keyword.
Here we insert *many* tuples into PRODUCT

INSERTION: AN EXAMPLE

Product(name, listPrice, category)
Purchase(prodName, buyerName, price)

prodName is foreign key in Product.name

Suppose database got corrupted and we need to fix it:

Product

name	listPrice	category
gizmo	100	gadgets

Purchase

prodName	buyerName	price
camera	John	200
gizmo	Smith	80
camera	Smith	225

Task: insert in Product all prodNames from Purchase

INSERTION: AN EXAMPLE

```
INSERT  INTO  Product(name)

 SELECT  DISTINCT  prodName
 FROM     Purchase
 WHERE   prodName  NOT IN (SELECT  name FROM  Product)
```

name	listPrice	category
gizmo	100	Gadgets
camera	-	-

INSERTION: AN EXAMPLE

```
INSERT  INTO  Product(name, listPrice)

 SELECT  DISTINCT  prodName, price
 FROM  Purchase
 WHERE  prodName  NOT IN (SELECT  name FROM  Product)
```

name	listPrice	category
gizmo	100	Gadgets
camera	200	-
camera ??	225 ??	-

← Depends on the implementation

DELETIONS

Example:

```
DELETE  FROM  PURCHASE

WHERE  seller = 'Joe'  AND
         product = 'Brooklyn Bridge'
```

Factoid about SQL: there is no way to delete only a single occurrence of a tuple that appears twice in a relation.

UPDATES

Example:

```
UPDATE   PRODUCT
SET      price = price/2
WHERE  Product.name  IN
                (SELECT product
                 FROM     Purchase
                 WHERE  Date ='Oct, 25, 1999');
```

YOU RAN YOUR QUERY… WHAT HAPPENED?

The database program parses the SQL, then formulates a *query plan*

- By examining the scheme and knowledge about sizes and data access patterns, and checking for existing indices, the DBMS first enumerates all possible execution sequences
- Now using those size and access pattern predictions, it assigns a cost in each case: an estimate, but one it can refine as it runs. More and more DBMS systems use machine learning at this step
- Finally, the query or update is executed

Notice that this is *much* more than a key-value store can offer!

ACID "VERSUS" STATE MACHINE REPLICATION?

In fact, the models are very similar!

Both take the view that the operation occurs as if the system was idle, and if data is replicated, all replicas see updates in the same order.

But SQL queries can be very complex and require many steps and execution stages. SMR updates are "one shot" actions like **put** that can be executed entirely as soon as the request reaches the replica.

CAN A KEY-VALUE SYSTEM SUPPORT FULL DATABASE TRANSACTIONS?

In fact, yes, but this is tricky to do correctly

We are exploring a way to extend Cascade to have this option as a built-in feature, but we think of it as a research project – an experiment.

In commercial settings, key-value stores are NoSQL system: they “speak” a subset of SQL, but lack the ACID guarantees of a true database

SUMMARY

SQL is a powerful tool in big data settings. Very flexible, fast. Helpful to make queries easy to optimize, but in fact modern databases are smart and should find an optimal way to run your logic.

Key-value stores don't support the full SQL model, but might offer some of the same API elements. We refer to this as the "NoSQL" subset

Both are extremely important in the cloud!