Module 7 - Review

Querying Data

Manos Papagelis

What is SQL?

- Declarative
  - Say “what to do” rather than “how to do it”
    - Avoid data-manipulation details needed by procedural languages
  - Database engine figures out “best” way to execute query
    - Called “query optimization”
    - Crucial for performance: “best” can be a million times faster than “worst”
- Data independent
  - Decoupled from underlying data organization
    - Views (= precomputed queries) increase decoupling even further
    - Correctness always assured… performance not so much
  - SQL is standard and (nearly) identical among vendors
    - Differences often shallow, syntactical

Fairly thin wrapper around relational algebra

SQL Main Components

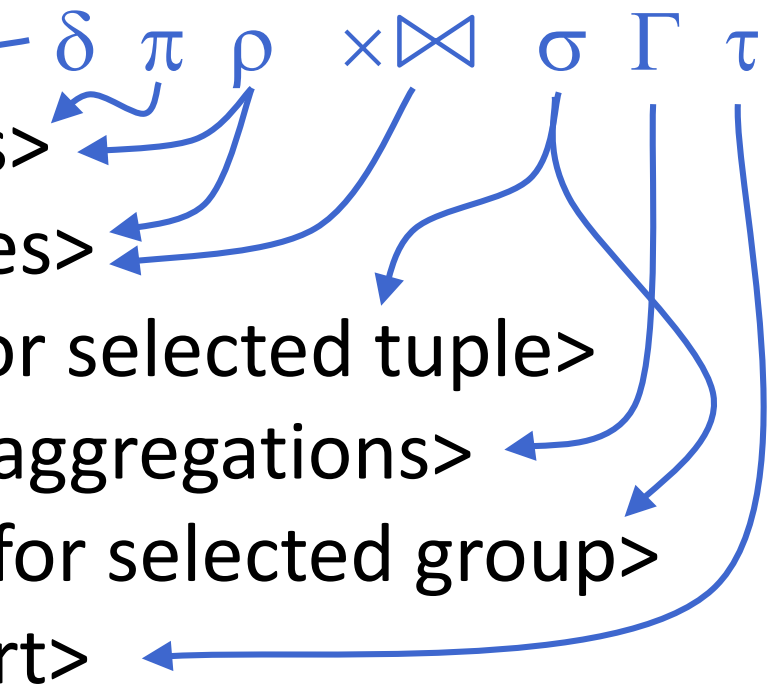
- Queries
  - Subset of SQL for read-only access to database
  - SELECT statements
- Data Definition Language (DDL)
  - Subset of SQL used to describe database schemas
  - CREATE, ALTER, DROP statements
  - Data types, Integrity constraints
- Data Manipulation Language (DML)
  - Subset of SQL used to manipulate data in databases
  - INSERT, DELETE, UPDATE statements
- Views and Indexes (Indices)

QUERIES

What does an SQL Query look like?

- Query syntax

  SELECT <desired attributes>
  FROM <one or more tables>
  WHERE<predicate holds for selected tuple>
  GROUP BY <key columns, aggregations>
  HAVING <predicate holds for selected group>
  ORDER BY <columns to sort>



Example

Orders

OID	OrderDate	OrderPrice	Customer
1	2008/11/12	1000	Hansen
2	2008/10/23	1600	Nilsen
3	2008/09/02	700	Hansen
4	2008/09/03	300	Hansen
5	2008/08/30	2000	Jensen
6	2008/10/04	100	Nilsen

Find if the customers "Hansen" or "Jensen“ have a total order of more than 1500

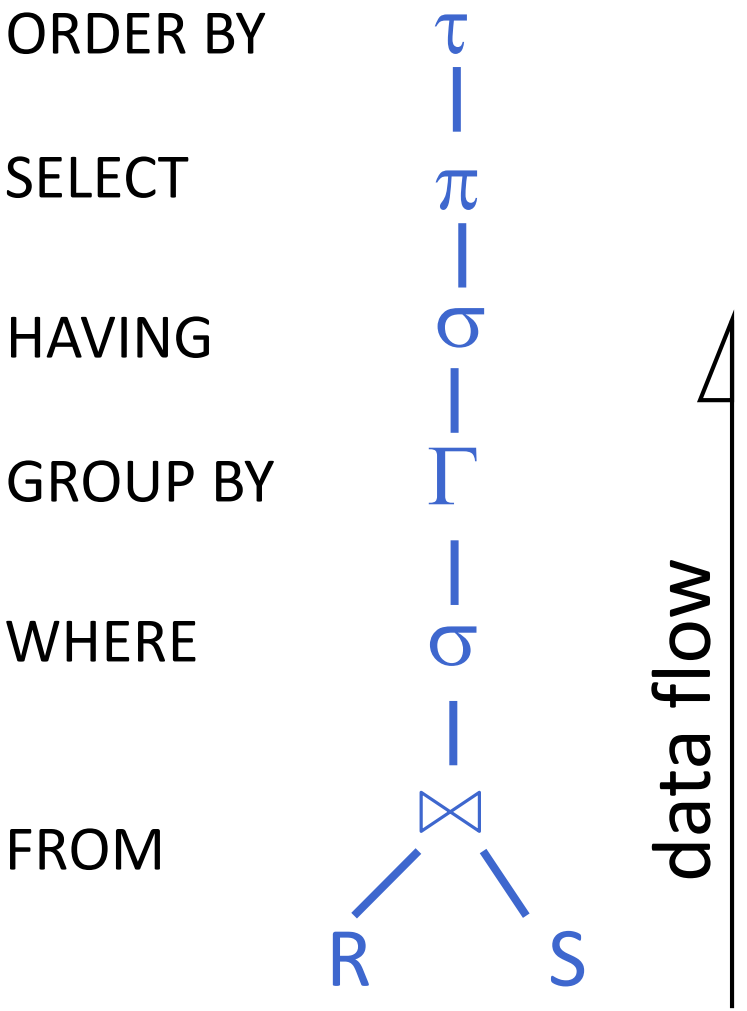
Query:

```
SELECT    Customer, SUM(OrderPrice) AS Total
FROM      Orders
WHERE     Customer = 'Hansen' OR Customer = 'Jensen'
GROUP BY  Customer
HAVING    SUM(OrderPrice) > 1500
ORDER BY  Customer DESC
```

Query Result:

Customer	Total
Jensen	2000
Hansen	2000

What does SQL *really* look like?



That's not so bad, is it?

WORKING EXAMPLES

Example Database

Employee(FirstName,Surname,Dept,Office,Salary,City)
Department(DeptName,Address,City)

Home city

EMPLOYEE

FirstName	Surname	Dept	Office	Salary	City
Mary	Brown	Administration	10	45	London
Charles	White	Production	20	36	Toulouse
Gus	Green	Administration	20	40	Oxford
Jackson	Neri	Distribution	16	45	Dover
Charles	Brown	Planning	14	80	London
Laurence	Chen	Planning	7	73	Worthing
Pauline	Bradshaw	Administration	75	40	Brighton
Alice	Jackson	Production	20	46	Toulouse

DEPARTMENT

DeptName	Address	City
Administration	Bond Street	London
Production	Rue Victor Hugo	Toulouse
Distribution	Pond Road	Brighton
Planning	Bond Street	London
Research	Sunset Street	San José

City of work

Example: Simple SQL Query

Employee(FirstName, Surname, Dept, Office, Salary, City)
Department(DeptName, Address, City)

"Find the salaries of employees named Brown"

SELECT Salary **AS** Remuneration

FROM Employee

WHERE Surname = 'Brown'

Result:

Remuneration
45
80

Example: Simple (Equi-)Join Query

Employee(FirstName, Surname, Dept, Office, Salary, City)
Department(DeptName, Address, City)

"Find the names of employees and their cities of work“

SELECT Employee.FirstName, Employee.Surname, Department.City
FROM Employee, Department
WHERE Employee.Dept = Department.DeptName

Result:

FirstName	Surname	City
Mary	Brown	London
Charles	White	Toulouse
Gus	Green	London
Jackson	Neri	Brighton
Charles	Brown	London
Laurence	Chen	London
Pauline	Bradshaw	London
Alice	Jackson	Toulouse

(alternative?)

Alternative (and more correct):
SELECT Employee.FirstName, Employee.Surname, Department.City
FROM Employee E **JOIN** Department D **ON** E.Dept = D.DeptName

Example: Predicate Conjunction

Employee(FirstName, Surname, Dept, Office, Salary, City)
Department(DeptName, Address, City)

"Find the first names and surnames of employees who work in office number 20 of the Administration department":

SELECT FirstName, Surname

FROM Employee

WHERE Office = '20' **AND** Dept = 'Administration'

Result:

FirstName	Surname
Gus	Green

Example: Operators **Sum**, **Avg**, **Max** and **Min**

Employee(FirstName, Surname, Dept, Office, Salary, City)
Department(DeptName, Address, City)

"Find the sum of all salaries for the Administration department":

SELECT sum(Salary) **AS** SumSalary
FROM Employee
WHERE Dept = 'Administration'

Result:

SumSalary
125

Example: GROUP BY

Employee(FirstName, Surname, Dept, Office, Salary, City)
Department(DeptName, Address, City)

"Find the sum of salaries of all the employees of each department":

SELECT Dept, sum(Salary) as TotSal
FROM Employee
GROUP BY Dept

Result:

Dept	TotSal
Administration	125
Distribution	45
Planning	153
Production	82

Example: GROUP BY Semantics

GROUP BY Processing:

- the query is executed without **GROUP BY** and without aggregate operators

  SELECT Dept, Salary as TotSal

  FROM Employee
- … then the query result is divided in subsets characterized by the same values for the **GROUP BY** attributes (in this case, Dept):
- Finally, the aggregate operator **sum** is applied separately to each group

Dept	Salary
Administration	45
Production	36
Administration	40
Distribution	45
Planning	80
Planning	73
Administration	40
Production	46

Dept	Salary
Administration	45
Administration	40
Administration	40
Distribution	45
Planning	80
Planning	73
Production	36
Production	46

Dept	TotSal
Administration	125
Distribution	45
Planning	153
Production	82

Example: HAVING

Employee(FirstName, Surname, Dept, Office, Salary, City)
Department(DeptName, Address, City)

"Find which departments spend more than 100 on salaries":

SELECT Dept
FROM Employee
GROUP BY Dept
HAVING sum(Salary) > 100

Result:

Dept
Administration
Planning

‘HAVING’ clause

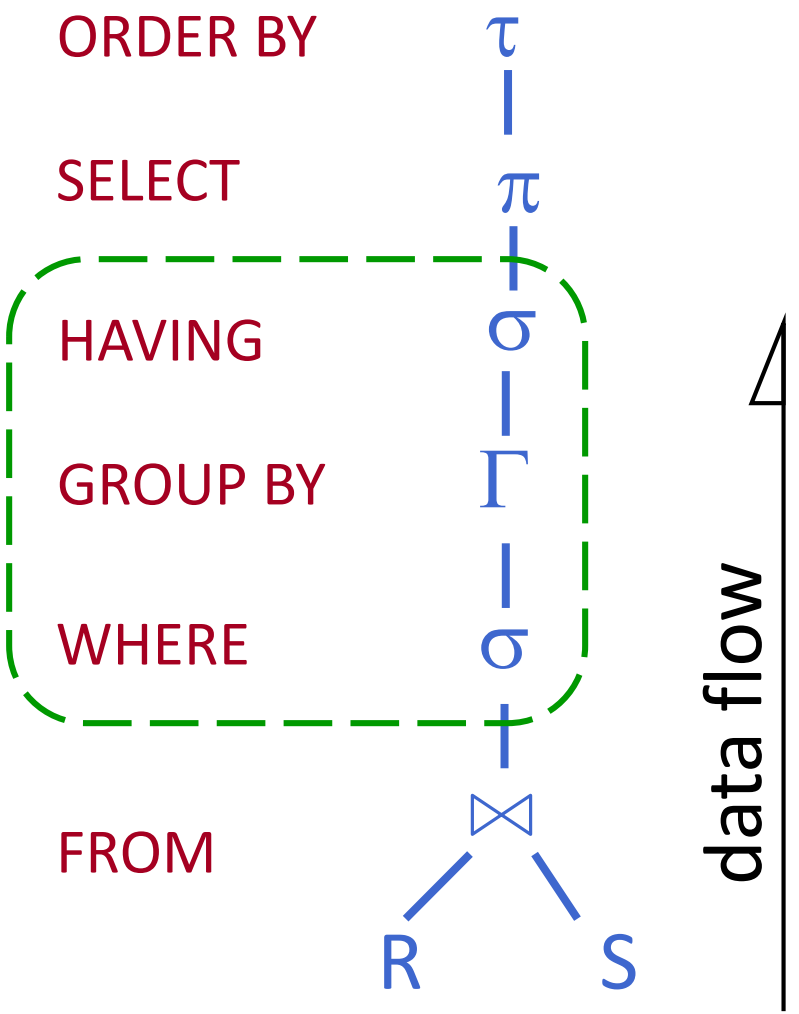
- Allows predicates on aggregate values
  - Groups which do not match the predicate are eliminated

  => **HAVING** is to groups what **WHERE** is to tuples
- Order of execution
  - WHERE is before GROUP BY

  => Aggregates not yet available when WHERE clause runs
  - GROUP BY is before HAVING

  => Scalar attributes still available

- In tree form:



DATA DEFINITION LANGUAGE (DDL)

Creating (Declaring) a Relation/Table

- To create a schema:

  CREATE SCHEMA schemaname [AUTHORIZATION user] ;

- To create a relation:

  CREATE TABLE <name> (

  <list of elements>

  );

- To delete a relation:

  DROP TABLE <name>;

- To alter a relation (add/remove column):

  ALTER TABLE <name> **ADD (DROP)** <element>

Kinds of Constraints

- Keys
- Foreign-key or referential-integrity constraints
  - Inter-relation constraints
- Value-based constraints
  - Constrain values of a particular attribute
- Tuple-based constraints
  - Relationship among components
- Assertions

Example: Primary and Foreign Key

```
CREATE TABLE Beers (
    name        CHAR(20) PRIMARY KEY,
    manf        CHAR(20) );

CREATE TABLE Sells (
    bar         CHAR(20),
    beer        CHAR(20),
    price       REAL,
    FOREIGN KEY(beer) REFERENCES  Beers(name));
```

DATA MANIPULATION LANGUAGE (DML)

Data Manipulation Language (DML)

- Syntax elements used for inserting, deleting and updating data in a database
- Modification statements include:
  - **INSERT** - for inserting data in a database
  - **DELETE** - for deleting data in a database
  - **UPDATE** - for updating data in a database
- All modification statements operate on *a set* of tuples (no duplicates)

VIEWS

Views

- A view is a relation defined in terms of stored tables (called base tables) and other views.
- Two kinds:
  - Virtual = not stored in the database; just a query for constructing the relation

    CREATE VIEW <name> **AS** <query>;
  - Materialized = actually constructed and stored

    CREATE MATERIALIZED VIEW <name> **AS** <query>;

Example

Beers(name, manf)

Bars(name, addr, license)

Drinkers(name, addr, phone)

Likes(drinker, beer)

Sells(bar, beer, price)

Frequents(drinker, bar)

Example: View Definition

CanDrink(drinker, beer) is a view "containing" the drinker-beer pairs such that the drinker frequents at least one bar that serves the beer:

```
CREATE VIEW CanDrink AS
    SELECT drinker, beer
    FROM Frequents, Sells
    WHERE Frequents.bar = Sells.bar;
```

Example: Accessing a View

Query a view as if it were a base table:

```
SELECT beer
FROM CanDrink
WHERE drinker = 'Sally';
```

Notes on Views

- Data independence (hide schema from apps)
  - DB team splits CustomerInfo into Customer and Address
  - View accomodate changes with web apps
- Data hiding (access data on need-to-know basis)
  - Doctor outsources patient billing to third party
  - View restricts access to billing-related patient info
- Code reuse
  - Very similar subquery appears multiple times in a query
  - View shortens code, improves readability, reduces bugs, …
  - Bonus: query optimizer often does a better job!

Example: Views and Queries

Employee(RegNo,FirstName,Surname,Dept,Office,Salary,City)

Department(DeptName,Address,City)

"Find the department with highest salary expenditures" (without using a view):

```
SELECT Dept
FROM Employee
GROUP BY Dept
HAVING sum(Salary) >= ALL  (
        SELECT sum(Salary) FROM Employee GROUP BY Dept)
```

Example: Views and Queries (cont.)

"Find the department with highest salary expenditures" (using a view):

```
CREATE VIEW SalBudget (Dept, SalTotal) AS
SELECT Dept, sum(Salary)
FROM Employee
GROUP BY Dept

SELECT Dept
FROM SalBudget
WHERE SalTotal = (SELECT max(SalTotal) FROM SalBudget)
```

Updates on Views

- Generally, it is impossible to modify a virtual view because it doesn't exist
- Can't we "translate" updates on views into "equivalent" updates on base tables?
  - Not always (in fact, not often)
  - Most systems prohibit most view updates

Materialized Views

- **Problem**: each time a base table changes, the materialized view may change
  - Cannot afford to recompute the view with each change
- Solution: Periodic reconstruction of the materialized view, which is otherwise "out of date"

Example: A Data Warehouse

- Wal-Mart stores every sale at every store in a database
- Overnight, the sales for the day are used to update a data warehouse = materialized views of the sales
- The warehouse is used by analysts to predict trends and move goods to where they are selling best

INDEXES (INDICES)

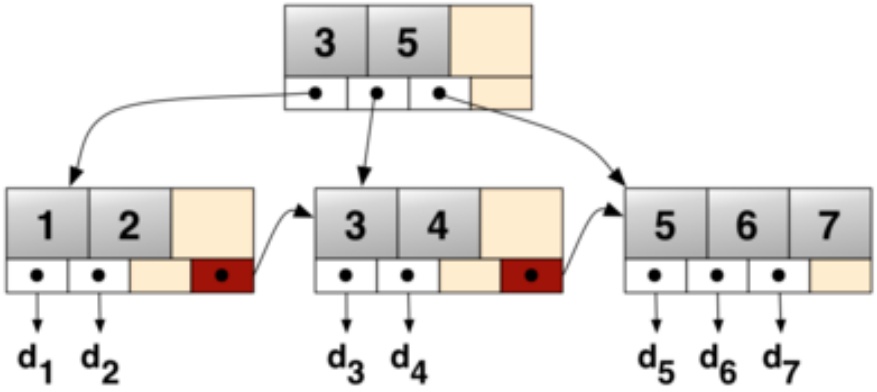
Index

- **Problem**: needle in haystack
  - Find all phone numbers with first name ‘Mary’
  - Find all phone numbers with last name ‘Li’
- **Index**: auxiliary database structure which provides random access to data
  - Index a set of attributes. No standard syntax! Typical is:

    CREATE INDEX indexName **ON** TableName(AttributeList);
  - Random access to any indexed attribute

    (e.g., retrieve a single tuple out of billions in **<5** disk accesses)
  - Similar to a hash table, but in a DBMS it is a **balanced search tree** with giant nodes (a full disk page) called a *B-tree*

B+ Tree Index



- Most common index type
- Key = indexed value
  - primary index
  - secondary index
- Value = record location
- *O(log n)* search
- *O(log n)* insert
- *O(n)* space

Example: Using Index

SELECT fname

FROM people

WHERE lname = ‘Papagelis‘

- **Without an index**:

  The DBMS must look at the *lname* column on every row in the table (this is known as a full table scan)
- **With an index** (defined on attribute lname):

  The DBMS simply follows the B-tree data structure until the ‘Papagelis’ entry has been found

This is much less computationally expensive than a full table scan

Another Example: Using Index

CREATE INDEX BeerInd **ON** Beers(manf);

CREATE INDEX SellInd **ON** Sells(bar, beer);

Query: Find the prices of beers manufactured by Pete's and sold by Joe's bar

SELECT price **FROM** Beers, Sells

WHERE manf = 'Pete''s' **AND** Beers.name = Sells.beer

AND bar = 'Joe''s Bar';

DBMS uses:

- **BeerInd** to get all the beers made by Pete's **fast**
- **SellInd** to get prices of those beers, with bar = 'Joe''s Bar' **fast**

Database Tuning

- How to make a database run fast?
  - Decide which indexes to create
- **Pro**: An index speeds up queries that can use it
- **Con**: An index slows down all modifications on its relation as the index must be modified too

Example: Database Tuning

- Suppose the only things we did with our beers database was:
  - Insert new beers into a relation (10%).
  - Find the price of a given beer at a given bar (90%).
- Then
  - **SellInd** on Sells(bar, beer) would be wonderful
  - **BeerInd** on Beers(manf) would be harmful

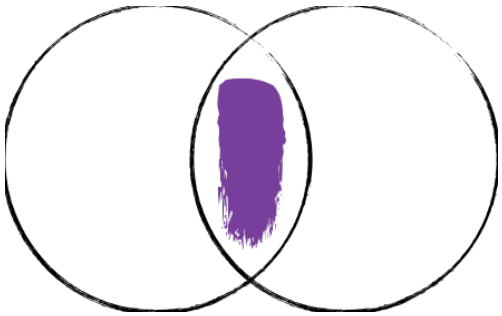
Make common case fast

Tuning Advisors

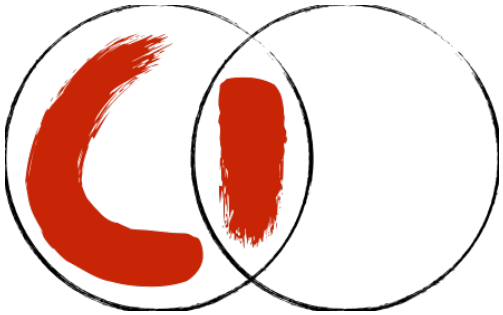
- A major research thrust
  - Because hand tuning is so hard
- An advisor gets a query load, e.g.:
  - Choose random queries from the history of queries run, or
  - Designer provides a sample workload
- The advisor generates candidate indexes and evaluates each on the workload
  - Feed each sample query to the query optimizer, which assumes only this one index is available
  - Measure the improvement/degradation in the average running time of the queries.

OTHER CONCEPTS

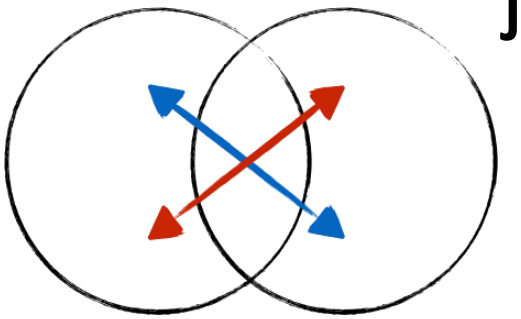
Joins



Inner Join



Left Outer Join



Cross Join

Query Optimization

If interested, see:

- db-book-ch13-query-optimization.pdf (in files)
- readings

Analytical SQL and Windows

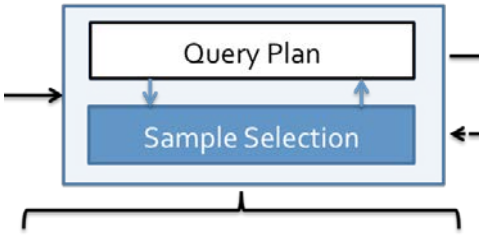
- Basic SQL doesn't describe all possible queries
- Think of
  - Time series analysis
  - Rolling sums
  - Ranking rows relative to each other
- Queries that have inter-row dependencies require
  - **Partition** operator to define **windows**
  - **Rank** operator to rank tuples inside a **window**

Approximate Query

- As data grows, query times increase
- Approximation can yield results in constant time
  - Trade-off: results are within a margin of error
- Data Sketching
  - Captures dynamics well
  - Only applicable to certain operations
- Uniform + Stratified Sampling
  - Broadly applicable
  - Greater storage burden

BlinkDB: Stratified Query System

SELECT sum(a), **avg**(b)
FROM T
WITHIN 2s;



(198.0+-21.2, 19.1+-0.8)