SQL

Database Systems: The Complete Book
Ch 2.3, 6.1-6.4

Project Outline

SQL Query

.sql

Parser & Translator

JAR JSqlParser

???

Relational Algebra

???

Hope and Duct Tape?

Query Result

SQL is Human Readable

- Lots of Syntactic Sugar
  - WHERE vs HAVING
- Lots of Corner Cases
  - SELECT A, B vs SELECT A, SUM(B)
- Non-obvious evaluation strategy
  - SELECT … FROM R, S, T, … WHERE …

SQL is hard to evaluate directly!

Relational Algebra

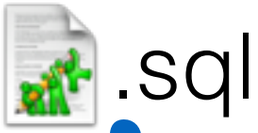
- Equivalent to SQL (to be discussed)
- SIMPLE! (only a handful of operators)
- “Non-declarative” (easy to rewrite)
- Minimal corner cases or syntactic sugar

“RA” is easier to interpret!

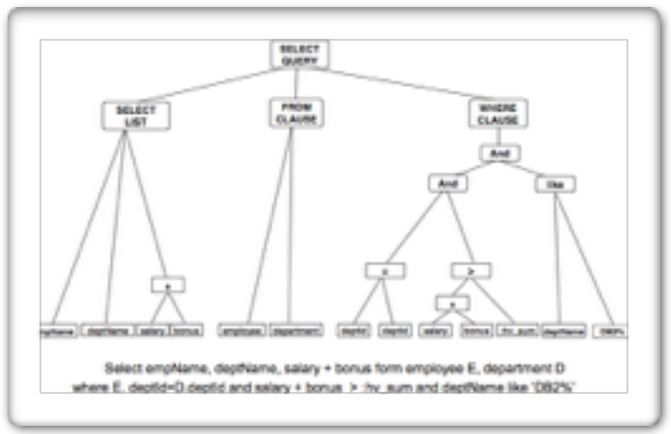
Relational Algebra

- Basic Relational Operators
  - Select (σ), Project (π), Cross/Join (x/⋈), Union (U), Relation (R, S, T, …), Minus (-)
- Extended Relational Operators (more next week)
  - Aggregates (`SUM,COUNT,MIN/MAX,AVERAGE`)
  - List Operators: Sort, Limit

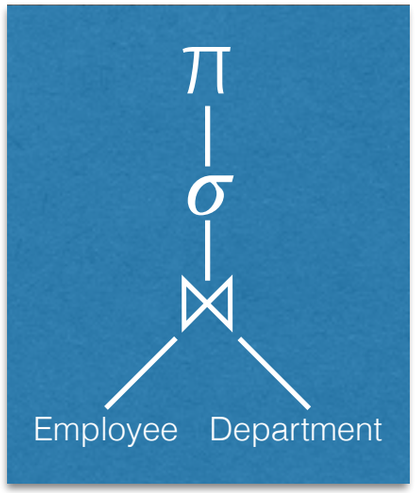
The Evaluation Pipeline


.sql

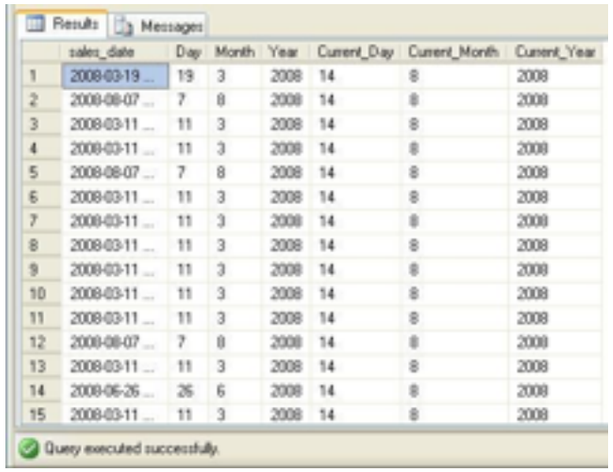
How does this work?
(later today)



Parsed Query




Data



Results

What does this look like?
(today)

How does this work?
(next class)

SQL

- Developed by IBM (for System R) in the 1970s.
- Standard used by many vendors.
  - SQL-86 (original standard)
  - SQL-89 (minor revisions; integrity constraints)
  - SQL-92 (major revision; basis for modern SQL)
  - SQL-99 (XML, window queries, generated default values)
  - SQL 2003 (major revisions to XML support)
  - SQL 2008 (minor extensions)
  - SQL 2011 (minor extensions; temporal databases)

A Basic SQL Query

(optional) keyword indicating that the answer should **not** contain duplicates

```
SELECT  [DISTINCT] target-list
```

A list of attributes of relations in relation-list

```
FROM    relation-list
```

A list of relation names
(possibly with a range-variable after each name)

```
WHERE   condition
```

Comparisons (‘=’, ‘<>’, ‘<‘, ‘>’, ‘<=’, ‘>=’) and other boolean predicates,
combined using AND, OR, and NOT
(a boolean formula)

A Basic SQL Query

net.sf.jsqlparser.statement.select.**PlainSelect**

SELECT [DISTINCT] target-list

mySelect.**getDistinct()**

mySelect.**getSelectItems()**

FROM relation-list

mySelect.**getFromItem()** and mySelect.**getJoins()**

WHERE condition

mySelect.**getWhere()**

Query Evaluation

```
SELECT   [DISTINCT] target-list
FROM     relation-list
WHERE    condition
```

1) Compute the 2^n combinations of tuples in all relations appearing in relation-list

2) Discard tuples that fail the condition

3) Delete attributes not in target-list

4) If DISTINCT is specified, eliminate duplicate rows

This is the least efficient strategy to compute a query!
A good optimizer will find more efficient strategies to compute **the same answer**.

DISTINCT

Why do you explicitly indicate that you want duplicate elimination in SQL?

Example-Wildcards

Find all officers on the Enterprise (Ship 1701A)

‘`*`’ denotes all attributes
‘`Officers.*`’ denotes all attributes in Officers

```
SELECT *
FROM Officers
WHERE Ship = ‘1701A’
```

FirstName,	LastName,	Ship
[James,	Kirk,	1701A]
[Leonard,	McCoy,	1701A]
[Spock,	SonOfSarek,	1701A]
. . .		

FirstName,	LastName,	Ship
[James,	Kirk,	1701A]
[Leonard,	McCoy,	1701A]
[Spock,	SonOfSarek,	1701A]
. . .		

net.sf.jsqlparser.statement.select.**AllColumns**
net.sf.jsqlparser.statement.select.**AllTableColumns**

Example-Condition

Find all officers on the Enterprise (Ship 1701A)

```
SELECT *
FROM Officers
 WHERE Ship = ‘1701A’
```

FirstName,	LastName,	Ship
[James,	Kirk,	1701A]
[Leonard,	McCoy,	1701A]
[Spock,	SonOfSarek,	1701A]
[Montgomery,	Scott,	1701A]
[Pavel,	Chekov,	1701A]
[Nyota,	Uhura,	1701A]

FirstName,	LastName,	Ship
[James,	Kirk,	1701A]
[Leonard,	McCoy,	1701A]
[Spock,	SonOfSarek,	1701A]
[Montgomery,	Scott,	1701A]
[Hikaru,	Sulu,	2000]
[Pavel,	Chekov,	1701A]
[Nyota,	Uhura,	1701A]
[Christine,	Chapel,	0001]

Example-Target List

Find just **names** of all officers on the Enterprise

```
SELECT O.FirstName,O.LastName
FROM Officers O
WHERE O.Ship = ‘1701A’
```

FirstName,	LastName
[James,	Kirk]
[Leonard,	McCoy]
[Spock,	SonOfSarek]
[Montgomery,	Scott]
[Pavel,	Chekov]
[Nyota,	Uhura]

FirstName,	LastName,	Ship
[James,	Kirk,	1701A]
[Leonard,	McCoy,	1701A]
[Spock,	SonOfSarek,	1701A]
[Montgomery,	Scott,	1701A]
[Hikaru,	Sulu,	2000]
[Pavel,	Chekov,	1701A]
[Nyota,	Uhura,	1701A]
[Christine,	Chapel,	0001]

Example-Multiple Relations

In English, what does this query compute?

```
SELECT FirstName,LastName
FROM Officers, Ships
WHERE Ship = ID
AND Location = ‘Vulcan’
```

FirstName,	LastName,	Ship
[James,	Kirk,	1701A]
[Leonard,	McCoy,	1701A]
[Spock,	SonOfSarek,	1701A]
[Montgomery,	Scott,	1701A]
[Hikaru,	Sulu,	2000]
[Pavel,	Chekov,	1701A]
[Nyota,	Uhura,	1701A]
[Christine,	Chapel,	0001]

ID,	Name,	Location
[1701A,	Enterprise-A,	Andoria]
[2000,	Excelsior,	Vulcan]
[1864,	Reliant,	Ceti Alpha VI]

FirstName,	LastName
[Hikaru,	Sulu]

Who is on a ship located at Vulcan?

Example-Multiple Relations

```
SELECT FirstName,LastName FROM Officers, Ships
WHERE Ship = ID AND Location = ‘Vulcan’
```

mySelect.**getFromItem()** returns

```
  ….schema.Table(Officers)
```

mySelect.**getJoins()** returns

```
  List(
    ….select.Join(Table(Ships), {simple})
  )
```

Range Variables

```
SELECT FirstName,LastName
FROM   Officers, Ships
WHERE  Ship = ID
  AND  Location = 'Vulcan'
```

is the same as

```
SELECT Officers.FirstName,Officers.LastName
FROM   Officers, Ships
WHERE  Officers.Ship = Ships.ID
  AND  Ships.Location = 'Vulcan'
```

But it's good style to use range variables and fully-qualified attribute names!

is the same as

```
SELECT O.FirstName,O.LastName
FROM   Officers O, Ships S
WHERE  O.Ship = S.ID
  AND  S.Location = 'Vulcan'
```

JSqlParser calls this an "alias"

Expressions

```
SELECT O.age,
       age1 = O.age*0.2,
       O.age*3.0 AS age2
FROM   Officers O
```

↓

[age, age1, age2]

Arithmetic expressions can appear in targets or conditions.
Use ‘=’ or ‘AS’ to assign names to these attributes.
(The behavior of unnamed attributes is unspecified)

Strings

```
SELECT O.FirstName, O.LastName
FROM   Officers O
WHERE  S.LastName LIKE ‘Ch%e%’
```

```
[Pavel,      Chekov]
[Christine,  Chapel]
```

SQL uses single quotes for `‘string literals’`

Strings

```
SELECT O.FirstName, O.LastName
FROM   Officers O
WHERE  O.LastName LIKE ‘Ch%e%’
```

↓

```
[Pavel,      Chekov]
[Christine,  Chapel]
```

`LIKE` is used for String Matches

‘**`%`**’ matches 0 or more characters

(like RegEx **`/.*/`**)

Strings

```
SELECT O.FirstName, O.LastName
FROM   Officers O
WHERE  O.LastName LIKE 'Ch_%e%'
```

↓

~~[Pavel, Chekov]~~
[Christine, Chapel]

`LIKE` is used for String Matches

'`%`' matches 0 or more characters
(like RegEx `/.*/`)

UNION

Computes the **union** of any two **union-compatible sets** of tuples

```
SELECT O.FirstName
FROM Officers O
WHERE O.LastName = 'Kirk'
   OR O.LastName = 'Picard'
```

is the same as

```
SELECT O.FirstName FROM Officers O
WHERE O.LastName = 'Kirk'

UNION

SELECT O.FirstName FROM Officers O
WHERE O.LastName = 'Picard'
```

UNION

net.sf.jsqlparser.statement.select.**Union**

```
SELECT O.FirstName FROM Officers O
WHERE O.LastName = ‘Kirk’
UNION
SELECT O.FirstName FROM Officers O
WHERE O.LastName = ‘Picard’
```

myUnion.**getPlainSelects()**

Nested Queries

What does this query compute?

```
SELECT O.FirstName, O.LastName
FROM Officers O
WHERE O.ID IN (SELECT V.Officer
               FROM   Visited V
               WHERE  V.Planet = 'Vulcan')
```

Use **`NOT IN`** for all officers who have never visited 'Vulcan'

`IN` nested query must have exactly **one** attribute

`net.sf.jsqlparser.expression.operators.relational.`**`InExpression`**

`net.sf.jsqlparser.statement.select.`**`SubSelect`**

Nested Queries

(With Correlation)

```
SELECT O.FirstName, O.LastName
FROM Officers O
WHERE EXISTS (SELECT *
              FROM   Visited V
              WHERE  V.Planet = 'Vulcan'
                AND  O.ID = V.Officer)
```

`EXISTS` is true if the nested query returns at least one result

The nested query can refer to attributes from the outer query

`net.sf.jsqlparser.expression.operators.relational.`**`ExistsExpression`**

More Set Operators

`IN` ⟶ `NOT IN`

`EXISTS` ⟶ `NOT EXISTS`

More Set Operators

`[op] ANY` `[op] ALL`

```
SELECT * FROM Officers O
WHERE O.Rank > ALL (SELECT O2.rank
                    FROM Officers O2,
                         Ships S
                    WHERE O2.Ship = S.ID
                      AND S.Name = 'Enterprise'
                   )
```

What does this compute?

Which officers outrank every officer on the Enterprise?

`net.sf.jsqlparser.expression.`**`AllComparisonExpression`**

From-Nesting

```
SELECT *
FROM Officers O,
     (SELECT V.Officer
      FROM Visited V
      WHERE V.Planet = ‘Andoria’
     ) A
WHERE O.ID = A.Officer
```

Queries are relations!

`net.sf.jsqlparser.statement.select.`**`SubSelect`**

Aggregate Operators

```
SELECT COUNT(*)
FROM Officers O, Ships S
WHERE O.Ship = S.ID
  AND S.Name = 'Enterprise'
```

What does this compute?

How many officers are on the Enterprise?

`net.sf.jsqlparser.expression.`**`Function`**

Aggregate Operators

```
COUNT(*)

COUNT(DISTINCT A[, B[, …]])

SUM([DISTINCT] A)

AVG([DISTINCT] A)

MAX(A)

MIN(A)
```

Group Exercise

```
SELECT * FROM Officers O
WHERE O.Rank > ALL (SELECT O2.rank
                    FROM Officers O2,
                         Ships S
                    WHERE O2.Ship = S.ID
                      AND S.Name = 'Enterprise'
                    )
```

How could you write this query without **ALL**?

Aggregate Operators

This query is illegal!

Why?

```
SELECT S.Name, AVG(O.Age)
FROM Officers O, Ships S
WHERE O.Ship = S.ID
GROUP BY S.Name
```

Grouping allows us to apply aggregates to groups of tuples.

Can't combine Aggregate and Non-Aggregate targets!

Group-By Queries

```
SELECT [DISTINCT] target-list
FROM relation-list
WHERE condition
GROUP BY grouping-list
HAVING group-condition
```

The target-list now contains
(a) grouped attributes
(b) aggregate expressions

Targets of type (a) must be a **subset** of the grouping-list

(intuitively each answer tuple corresponds to a single group,
and each group must have a single value for each attribute)

Group-By Queries

```
SELECT [DISTINCT] target-list
FROM relation-list
WHERE condition
GROUP BY grouping-list
HAVING group-condition
```

The condition is applied before grouping
The having-condition is applied after grouping

Group-By Queries

```
SELECT [DISTINCT] target-list
FROM relation-list
WHERE condition
GROUP BY grouping-list
HAVING group-condition
```

mySelect.**getHaving()**

mySelect.**getGroupByColumnReferences()**

Order By/Limit

How can we compute the Top 5 officers by rank?

```
SELECT O.Name, O.Rank
FROM Officers O
ORDER BY O.Rank
LIMIT 5
```

mySelect.**getOrderByElements()**

mySelect.**getLimit()**

Defining Relations in SQL

```
CREATE TABLE Officers
  ( FirstName CHAR(20),
    LastName  CHAR(20),
    Ship      CHAR(5),
    ID        INTEGER
  )

CREATE TABLE Ships
  ( ID        CHAR(5),
    Name      CHAR(20),
    Location  CHAR(40)
  )
```

The schema defines not only the column names, but also their types (domains)

For example a 20-character string

Modifying Relations

Destroy the relation ‘Officers’
All schema information AND tuples are deleted

```
DROP TABLE Officers
```

Add a new column (field) to the Ships relation
Every tuple in the current instance is extended with a ‘null’ value in the new field

```
ALTER TABLE Ships
   ADD COLUMN Commissioned DATE
```

Adding and Deleting Tuples

Insert single tuples using:

```
INSERT INTO Officers (FirstName, LastName, Ship)
  VALUES ('Benjamin', 'Sisko', '74205')
```

Can delete all tuples satisfying some condition (e.g., Ship = 2000)

```
DELETE FROM Officers O
  WHERE O.Ship = '2000'
```

More powerful data manipulation commands are available in SQL
(We'll discuss them later in the course)

SQL

- SQL is a language for querying relations
  - **`SELECT`** to access (query) data
    - Different features for different access patterns.
  - **`INSERT INTO`**, **`DELETE FROM`** to modify data
  - **`CREATE TABLE`**, **`DROP TABLE`**, **`ALTER TABLE`** to modify relations
- Next time…
  - Translating SQL to Relational Algebra (equivalence)