Database Systems
CSE 514

Lectures 06
Size Estimation; NoSQL, JSon
Today

• Database statistics and size estimation

• NoSQL and the semistructured data model
Query Evaluation Steps

1. **Parse & Check Query**
   - Translate query string into internal representation
   - Check syntax, access control, table names, etc.

2. **Decide how best to answer query: query optimization**
   - Logical plan → physical plan

3. **Query Execution**

4. **Return Results**

DATA514 - Winter 2018
Database Statistics

- Collect statistical summaries of stored data
- Estimate size (=cardinality), bottom-up
- Estimate cost by using the estimated size
Database Statistics

• Number of tuples $T(R) = \text{cardinality}$

• Number of distinct values of attribute $a$ $V(R,a)$

• Other statistics (later)

Collection approach: periodic, using sampling
Size Estimation Problem

\[
S = \text{SELECT} \ *
\]
\[
\text{FROM} \quad R1, \ldots, \ Rn
\]
\[
\text{WHERE} \quad \text{cond}_1 \ \text{AND} \ \text{cond}_2 \ \text{AND} \ \ldots \ \text{AND} \ \text{cond}_k
\]

Given \( T(R1), T(R2), \ldots, T(Rn) \)

Estimate \( T(S) \)

How can we do this? Note: doesn’t have to be exact.
Size Estimation Problem

\[
S = \text{SELECT } * \\
\text{FROM R1, ..., Rn} \\
\text{WHERE } \text{cond}_1 \text{ AND cond}_2 \text{ AND } \ldots \text{ AND cond}_k
\]

Remark: \( T(S) \leq T(R1) \times T(R2) \times \ldots \times T(Rn) \)
Selectivity Factor

• Each condition $\text{cond}$ reduces the size by some factor called \textit{selectivity factor}.

• Assuming independence, multiply the selectivity factors.
Example

```
SELECT *
FROM R, S, T
WHERE R.B=S.B and S.C=T.C and R.A<40
```

T(R) = 30k, T(S) = 200k, T(T) = 10k

Selectivity of $R.B = S.B$ is $1/3$
Selectivity of $S.C = T.C$ is $1/10$
Selectivity of $R.A < 40$ is $\frac{1}{2}$

What is the estimated size of the query output?
Example

\[ \text{SELECT } * \text{ FROM R, S, T WHERE R.B=S.B and S.C=T.C and R.A<40} \]

T(R) = 30k, T(S) = 200k, T(T) = 10k

Selectivity of R.B = S.B is 1/3
Selectivity of S.C = T.C is 1/10
Selectivity of R.A < 40 is \( \frac{1}{2} \)

What is the estimated size of the query output?

\[ 30k \times 200k \times \frac{1}{3} \times \frac{1}{10} \times \frac{1}{2} = 1TB \]
Statistical Model

What is the probability space?

\[ S = \text{SELECT list} \]
\[ \text{FROM } R_1 \text{ as } x_1, \ldots, R_k \text{ as } x_k \]
\[ \text{WHERE Cond} \quad -- \text{a conjunction of predicates} \]
What is the probability space?

\[ S = \text{SELECT list} \]
\[ \text{FROM } R_1 \text{ as } x_1, \ldots, R_k \text{ as } x_k \]
\[ \text{WHERE } \text{Cond} -- \text{a conjunction of predicates} \]

\[(x_1, x_2, \ldots, x_k), \text{ drawn randomly, independently from } R_1, \ldots, R_k\]

\[ \Pr(R_1.A = 40) = \text{prob. that random tuple in } R_1 \text{ has } A=40 \]

\[ \Pr(R_1.A = 40 \text{ and } J_{R_1.B = R_2.C} \text{ and } R_2.D = 90) = \text{prob. that } \ldots \]

\[ \mathbb{E}[ |\text{SELECT } \ldots \text{ WHERE Cond}| ] = \Pr(\text{Cond}) \ast T(R_1) \ast T(R_2) \ast \ldots \ast T(R_k) \]
Statistical Model

What is the probability space?

\[ S = \text{SELECT list} \]
\[ \text{FROM} \quad R_1 \text{ as } x_1, \ldots, R_k \text{ as } x_k \]
\[ \text{WHERE} \quad \text{Cond} -- \text{a conjunction of predicates} \]

Three simplifying assumptions

**Uniform:** \( \Pr(R_1.A = 'a') = 1/V(R_1, A) \)

**Attribute Indep.:** \( \Pr(R_1.A = 'a' \text{ and } R_1.B = 'b') = \Pr(R_1.A = 'a') \Pr(R_1.B = 'b') \)

**Join Indep.:** \( \Pr(R_1.A = 'a' \text{ and } J_{R_1.B = R_2.C}) = \Pr(R_1.A = 'a') \Pr(J_{R_1.B = R_2.C}) \)
Rule of Thumb

• If selectivities are unknown, then:
  selectivity factor = 1/10
  [System R, 1979]
Using Data Statistics

- **Condition is** $A = c$  /* value selection on R */
  - **Selectivity** = $1/V(R,A)$

- **Condition is** $A < c$  /* range selection on R */
  - **Selectivity** = $(c - \text{Low}(R, A))/(\text{High}(R,A) - \text{Low}(R,A))T(R)$

- **Condition is** $A = B$  /* $R \bowtie_{A=B} S$ */
  - **Selectivity** = $1 / \max(V(R,A),V(S,A))$
  - (will explain next)
Selectivity of Join Predicates

Assumption:

- *Containment of values*: if $V(R,A) \leq V(S,B)$, then the set of $A$ values of $R$ is included in the set of $B$ values of $S$

  - Note: this indeed holds when $A$ is a foreign key in $R$, and $B$ is a key in $S$
Selectivity of Join Predicates

Assume \( V(R,A) \leq V(S,B) \)

- Each tuple \( t \) in \( R \) joins with \( T(S) / V(S,B) \) tuple(s) in \( S \)
- Hence \( T(R \Join_{A=B} S) = T(R) \ T(S) / V(S,B) \)

In general: \( T(R \bowtie_{A=B} S) = T(R) \ T(S) / \max(V(R,A),V(S,B)) \)
Selectivity of Join Predicates

Example:

• $T(R) = 10000$, $T(S) = 20000$
• $V(R,A) = 100$, $V(S,B) = 200$
• How large is $R \bowtie_{A=B} S$ ?
Histograms

- Statistics on data maintained by the RDBMS
- Makes size estimation much more accurate (hence, cost estimations are more accurate)
Histograms

Employee(ssn, name, age)

\[ T(\text{Employee}) = 25000, \ V(\text{Employee, age}) = 50 \]
\[ \text{min(age)} = 19, \ \text{max(age)} = 68 \]

\[ \sigma_{\text{age}=48}(\text{Employee}) = ? \]
\[ \sigma_{\text{age}>28 \ \text{and} \ \text{age}<35}(\text{Employee}) = ? \]
Histograms

Employee(ssn, name, age)

T(Employee) = 25000, V(Employee, age) = 50
min(age) = 19, max(age) = 68

\[ \sigma_{age=48}(Employee) = ? \]
\[ \sigma_{age>28 \text{ and } age<35}(Employee) = ? \]

<table>
<thead>
<tr>
<th>Age:</th>
<th>0..20</th>
<th>20..29</th>
<th>30-39</th>
<th>40-49</th>
<th>50-59</th>
<th>&gt; 60</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tuples</td>
<td>200</td>
<td>800</td>
<td>5000</td>
<td>12000</td>
<td>6500</td>
<td>500</td>
</tr>
</tbody>
</table>
Histograms

Employee(ssn, name, age)

\[ T(\text{Employee}) = 25000, \ V(\text{Employee, age}) = 50 \]
\[ \text{min(age)} = 19, \ \text{max(age)} = 68 \]

\[ \sigma_{\text{age}=48}(\text{Employee}) = ? \quad \sigma_{\text{age}>28 \ and \ age<35}(\text{Employee}) = ? \]

<table>
<thead>
<tr>
<th>Age:</th>
<th>0..20</th>
<th>20..29</th>
<th>30-39</th>
<th>40-49</th>
<th>50-59</th>
<th>&gt; 60</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tuples</td>
<td>200</td>
<td>800</td>
<td>5000</td>
<td>12000</td>
<td>6500</td>
<td>500</td>
</tr>
</tbody>
</table>

Estimate = 1200
Estimate = \(1 \times 80 + 5 \times 500 = 2580\)
Types of Histograms

• How should we determine the bucket boundaries in a histogram?
Types of Histograms

• How should we determine the bucket boundaries in a histogram?

• Eq-Width
• Eq-Depth
• Compressed
• V-Optimal histograms
Employee(ssn, name, age)

Histograms

Eq-width:

<table>
<thead>
<tr>
<th>Age</th>
<th>0..20</th>
<th>20..29</th>
<th>30-39</th>
<th>40-49</th>
<th>50-59</th>
<th>&gt; 60</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tuples</td>
<td>200</td>
<td>800</td>
<td>5000</td>
<td>12000</td>
<td>6500</td>
<td>500</td>
</tr>
</tbody>
</table>

Eq-depth:

<table>
<thead>
<tr>
<th>Age</th>
<th>0..20</th>
<th>20..29</th>
<th>30-39</th>
<th>40-49</th>
<th>50-59</th>
<th>&gt; 60</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tuples</td>
<td>1800</td>
<td>2000</td>
<td>2100</td>
<td>2200</td>
<td>1900</td>
<td>1800</td>
</tr>
</tbody>
</table>

Compressed: store separately highly frequent values: (48,1900)
V-Optimal Histograms

- Defines bucket boundaries in an optimal way, to minimize the error over all point queries.
- Computed rather expensively, using dynamic programming.
- Modern databases systems use V-optimal histograms or some variations.
Discussion in Class

• Small number of buckets
  – Hundreds, or thousands, but not more
  – WHY?

• Not updated during database update, but recomputed periodically
  – WHY?
Multidimensional Histograms

Classical example:

SQL query:
```
SELECT ... FROM ... 
WHERE Person.city = 'Seattle' ...
```

User “optimizes” it to:

```
SELECT ... FROM ... 
WHERE Person.city = 'Seattle' 
and Person.state = 'WA'
```

Big problem! (Why?)
Multidimensional Histograms

- Store distributions on two or more attributes
- Curse of dimensionality: space grows exponentially with dimension
- In practice: only two dimensional histograms
The New Hipster: NoSQL
NoSQL Motivation

• Originally motivated by Web 2.0 applications

• Goal is to scale simple OLTP-style workloads to thousands or millions of users (in class: OLTP v.s. OLAP)

• Users are doing both updates and reads
What is the Problem?

• Single server DBMS are too small for Web data

• Solution: scale out to multiple servers

• This is hard for the *entire* functionality of DMBS

• NoSQL: reduce functionality for easier scale up
  – Simpler data model
  – Simpler transactions
SQLite:
- One data file
- One user
- One DBMS application

But only a limited number of scenarios work with such model
Client-Server

Client Applications

DATA514 - Winter 2018
Client-Server

- One server running the database
- Many clients, connecting via the ODBC or JDBC (Java Database Connectivity) protocol.
One server running the database

Many clients, connecting via the ODBC or JDBC (Java Database Connectivity) protocol
Client-Server

• One *server* that runs the DBMS (or RDBMS):
  – Your own desktop, or
  – Some beefy system, or
  – A cloud service (SQL Azure)
Client-Server

- **One server** that runs the DBMS (or RDBMS):
  - Your own desktop, or
  - Some beefy system, or
  - A cloud service (SQL Azure)

- **Many clients** run apps and connect to DBMS
  - Microsoft’s Management Studio (for SQL Server), or
  - `psql` (for postgres)
  - Some Java program or some C++ program
Client-Server

• One *server* that runs the DBMS (or RDBMS):
  – Your own desktop, or
  – Some beefy system, or
  – A cloud service (SQL Azure)

• Many *clients* run apps and connect to DBMS
  – Microsoft’s Management Studio (for SQL Server), or
  – `psql` (for *postgres*)
  – Some Java program (HW5) or some C++ program

• Clients “talk” to server using JDBC/ODBC protocol
3-Tiers DBMS Deployment

File 1

File 2

File 3

DB Server

Browser
3-Tiers DBMS Deployment

Connection (e.g., JDBC)

HTTP/SSL

Browser
3-Tiers DBMS Deployment

- **DB Server**
  - File 1
  - File 2
  - File 3

- **App+Web Server**
  - Connection (e.g., JDBC)

- **Browser**
- **HTTP/SSL**

Web-based applications
3-Tiers DBMS Deployment

DB Server

File 1
File 2
File 3

Connection (e.g., JDBC)

HTTP/SSL

App+Web Server

App+Web Server

App+Web Server
3-Tier DBMS Deployment

File 1
File 2
File 3

DB Server

Why don’t we replicate the DB server too?

Replicate App server for scaleup

Connection (e.g., JDBC)

HTTP/SSL

App+Web Server
App+Web Server
App+Web Server
Replicating the Database

• Much harder, because the state must be unique, in other words the database must act as a whole

• Two basic approaches:
  – Scale up through partitioning
  – Scale up through replication
Scale Through **Partitioning**

- Partition the database across many machines in a cluster
  - Database now fits in main memory
  - Queries spread across these machines
- Can increase throughput
- Easy for writes but reads become expensive!
Scale Through Replication

- Create multiple copies of each database partition
- Spread queries across these replicas
- Can increase throughput and lower latency
- Can also improve fault-tolerance
- Easy for reads but writes become expensive!

Some requests

Three replicas

Other requests
Data Models

Taxonomy based on data models:

- **Key-value stores**
  - e.g., Project Voldemort, Memcached
- **Document stores**
  - e.g., SimpleDB, CouchDB, MongoDB
- **Extensible Record Stores**
  - e.g., HBase, Cassandra
Key-Value Stores Features

- **Data model**: (key, value) pairs
  - Key = string/integer, unique for the entire data
  - Value = can be anything (very complex object)

- **Operations**
  - Get(key), Put(key, value)
  - Operations on value not supported

- **Distribution / Partitioning**
  - No replication: key k is stored at server h(k)
  - 3-way replication: key k stored at h1(k), h2(k), h3(k)

How does get(k) work? How does put(k, v) work?
Example

• How would you represent the Flights data as key, value pairs?

• Option 1: key=fid, value=entire flight record
Example

- How would you represent the Flights data as key, value pairs?
  - Option 1: key=fid, value=entire flight record
  - Option 2: key=date, value=all flights that day
Example

• How would you represent the Flights data as key, value pairs?

  • Option 1: key=fid, value=entire flight record
  • Option 2: key=date, value=all flights that day
  • Option 3: key=(origin,dest), value=all flights between

How does query processing work?
Key-Value Stores Internals

- Data remains in main memory
- One type of impl.: distributed hash table
- Most systems also offer a persistence option
- Others use replication to provide fault-tolerance
- Some offer ACID transactions others do not
Data Models

Taxonomy based on data models:

• **Key-value stores**
  – e.g., Project Voldemort, Memcached

• **Document stores**
  – e.g., SimpleDB, CouchDB, MongoDB

• **Extensible Record Stores**
  – e.g., HBase, Cassandra, PNUTS
Document Stores Features

• **Data model**: (key, document) pairs
  – Key = string/integer, unique for the entire data
  – Document = JSON, or XML

• **Operations**
  – Get/put document by key
  – Limited, non-standard query language on JSON

• **Distribution / Partitioning**
  – Entire documents, as for key/value pairs

We will discuss JSON today
Data Models

Taxonomy based on data models:

• **Key-value stores**
  – e.g., Project Voldemort, Memcached

• **Document stores**
  – e.g., SimpleDB, CouchDB, MongoDB

• **Extensible Record Stores**
  – e.g., HBase, Cassandra, PNUTS
Extensible Record Stores

• Based on Google’s BigTable

• Data model is rows and columns

• Scalability by splitting rows and columns over nodes

• HBase is an open source implementation of BigTable
JSon and Semistructured Data
Where We Are

• So far we have studied the *relational data model*
  – Data is stored in tables (=relations)
  – Queries are expressions in the relational calculus (or relational algebra, or datalog, or SQL…)

• Today: Semistructured data model
  – Popular formats today: XML, JSON, protobuf
JSON - Overview

- JavaScript Object Notation = lightweight text-based open standard designed for human-readable data interchange. Interfaces in C, C++, Java, Python, Perl, etc.

- The filename extension is `.json`.

We will emphasize JSON as semi-structured data.
JSon vs Relational

• Relational data model
  – Flat structure (tables)
  – Schema must be fixed in advanced
  – Binary representation: good for performance, bad for exchange
  – Query language based on Relational Calculus

• Semistructured data model / JSon
  – Flexible, nested structure (trees)
  – Does not require predefined schema ("self describing”)
  – Text representation: good for exchange, bad for performance
JSon Syntax

```json
{
  "book": [
    {
      "id": "01",
      "language": "Java",
      "author": "H. Javeson",
      "year": 2015
    },
    {
      "id": "07",
      "language": "C++",
      "edition": "second",
      "author": "E. Sepp",
      "price": 22.25
    }
  ]
}
```
JSon Terminology

• Data is represented in name/value pairs.
• Curly braces hold objects
  – Each object is a list of name/value pairs separated by , (comma)
  – Each pair is a name is followed by ':'(colon) followed by the value
• Square brackets hold arrays and values are separated by , (comma).
JSON Data Structures

• Collections of name-value pairs:
  – {“name1”: value1, “name2”: value2, …}
  – The “name” is also called a “key”

• Ordered lists of values:
  – [obj1, obj2, obj3, …]
Avoid Using Duplicate Keys

The standard allows them, but many implementations don’t

```json
{"id":"07",
 "title": "Databases",
 "author": "Garcia-Molina",
 "author": "Ullman",
 "author": "Widom"
}
```

```json
{"id":"07",
 "title": "Databases",
 "author": ["Garcia-Molina",
             "Ullman",
             "Widom"]
}
```
JSon Datatypes

- Number
- String = double-quoted
- Boolean = true or false
- nul/empty
JSON Semantics: a Tree!

```json
{ "person": [
  { "name": "Mary",
    "address": {
      "street": "Maple",
      "no": 345,
      "city": "Seattle"
    }
  },
  { "name": "John",
    "address": "Thailand",
    "phone": 2345678
  }
]
}
```
JSon Data

• JSon is self-describing
• Schema elements become part of the data
  – Relational schema: person(name, phone)
  – In Json “person”, “name”, “phone” are part of the data, and are repeated many times
• Consequence: JSon is much more flexible
• JSon = semistructured data
Mapping Relational Data to JSON

<table>
<thead>
<tr>
<th>Person</th>
<th>name</th>
<th>phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>3634</td>
<td></td>
</tr>
<tr>
<td>Sue</td>
<td>6343</td>
<td></td>
</tr>
<tr>
<td>Dirk</td>
<td>6363</td>
<td></td>
</tr>
</tbody>
</table>

```json
{"person": [{"name": "John", "phone": 3634}, {"name": "Sue", "phone": 6343}, {"name": "Dirk", "phone": 6363}]
```
## Mapping Relational Data to JSON

May inline foreign keys

### Person

<table>
<thead>
<tr>
<th>name</th>
<th>phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>3634</td>
</tr>
<tr>
<td>Sue</td>
<td>6343</td>
</tr>
</tbody>
</table>

### Orders

<table>
<thead>
<tr>
<th>personName</th>
<th>date</th>
<th>product</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>2002</td>
<td>Gizmo</td>
</tr>
<tr>
<td>John</td>
<td>2004</td>
<td>Gadget</td>
</tr>
<tr>
<td>Sue</td>
<td>2002</td>
<td>Gadget</td>
</tr>
</tbody>
</table>

```json
{"Person": [
    {
        "name": "John",
        "phone": 3646,
        "Orders": [
            {
                "date": 2002,
                "product": "Gizmo"
            },
            {
                "date": 2004,
                "product": "Gadget"
            }
        ]
    },
    {
        "name": "Sue",
        "phone": 6343,
        "Orders": [
            {
                "date": 2002,
                "product": "Gadget"
            }
        ]
    }
]}
```
**JSON = Semi-structured Data (1/3)**

- **Missing attributes:**

  ```json
  {
  "person":
  [{
  "name":"John",
  "phone":1234},
  {
  "name":"Joe"
  }]
  }
  no phone !
  ```

- **Could represent in a table with nulls**

<table>
<thead>
<tr>
<th>name</th>
<th>phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>1234</td>
</tr>
<tr>
<td>Joe</td>
<td>-</td>
</tr>
</tbody>
</table>
JSon = Semi-structured Data (2/3)

- Repeated attributes

{
  "person": [
    {
      "name": "John",
      "phone": 1234
    },
    {
      "name": "Mary",
      "phone": [1234, 5678]
    }
  ]
}

Two phones!

- Impossible in one table:

<table>
<thead>
<tr>
<th>name</th>
<th>phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mary</td>
<td>2345</td>
</tr>
</tbody>
</table>
JSon = Semi-structured Data (3/3)

- Attributes with different types in different objects

```json
{
  "person":
  [
    {
      "name":"Sue", "phone":3456,
      "name":{"first":"John","last":"Smith"},"phone":2345
    }
  ]
}
```

- Nested collections
- Heterogeneous collections