What happens inside?

```
SELECT *  
WHERE species = 'Giraffe'
```
What happens inside?

SELECT name, addr, balance
WHERE userid = ?

<table>
<thead>
<tr>
<th>name</th>
<th>addr</th>
<th>balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
What happens inside?

SELECT image, imgid
WHERE likelyPlanet(image) = TRUE

Database systems can support
• interactive queries
• web applications
• large-scale science
and many other workloads
What happens inside?
What happens inside?
What happens inside?

1. Client makes conn, xmits SQL
2. Admission control prevents too much incoming work
3. Check role-based privileges
4. Query gets assigned to a “worker”
5. How should you do the work?
   1. Process per worker
   2. Thread per worker
   3. Process pool
What happens inside?

- Admission Control
- Catalog Manager
  - Memory Manager
  - Administration, Monitoring & Utilities
  - Replication and Loading Services
  - Batch Utilities
  - Shared Components and Utilities (Section 7)
- Dispatch and Scheduling
  - Query Parsing and Authorization
  - Query Rewrite
  - Query Optimizer
  - Plan Executor
  - DDL and Utility Processing
- Process Manager (Section 2)
  - Relational Query Processor (Section 4)
  - Access Methods
  - Buffer Manager
  - Lock Manager
  - Log Manager
  - Transactional Storage Manager (Sections 5 & 6)
What happens inside?
What happens inside?
DB Core Components

Admission Control
Connection Management

Query System
- Parser
- Rewriter
- Planner
- Executor

Storage System
- Access Methods
- Buffer Manager
- Lock Manager
- Log Manager

Optimizer
Flow of a Query

Admission Control
Connection Management

Query System
- Parser
- Rewriter
- Planner
- Executor

SQL
- SELECT movieTitle
- FROM StarsIn, MovieStar
- WHERE starName = name
- AND birthday LIKE "%1990"

Parse Tree
- π_{movieTitle}
- σ_{birthday...}
- σ_{starName = name}

Optimization
- Planner
- Executor

Storage System
- Access Methods
- Buffer Manager
- Lock Manager
- Log Manager
Plan for Next Few Lectures

Today
+Lec 6
Lec 7
Lec 8 – Join Algos
Optimizer (Lec 10)

Storage System
Access Methods
Buffer Manager
Lock Manager
Log Manager

Query System
Parser
Rewriter
Planner
Executor

Admission Control
Connection Management
Query Processing Steps

- Admission Control
- Query Rewriting
- Plan Formulation (SQL → Tree)
- Optimization
Query Processing Steps

• Admission Control
• **Query Rewriting**
• Plan Formulation (SQL → Tree)
• Optimization
Query Rewriting

- View Substitution
- Predicate Transforms
- Subquery Flattening
View Substitution

emp : id, sal, age, dept

create view sals as ( 
    select dept, avg(sal) as sal 
    from emp 
    group by dept 
) 

select sal from sals where dept = 'eecs';

select sal from ( 
    select dept, avg(sal) as sal 
    from emp 
    group by dept 
) where dept = 'eecs';
Predicate Transforms

• Remove & simplify expressions, improve

• Constant Simplification
  \[ \text{WHERE } \text{sal} > 1000 + 4000 \implies \text{WHERE } \text{sal} > 5000 \]

• Exploit constraints
  \[ a.\text{did} = 10 \text{ and } a.\text{did} = \text{dept.dno} \text{ and } \text{dept.dno} = 10 \]

• Remove redundant expressions
  \[ a.\text{sal} \geq 10k \text{ and } \text{sal} > 20k \]
Subquery Flattening

• Many Subqueries Can Be Eliminated

```
select sal from (  
    select dept, avg(sal) as sal  
    from emp  
    group by dept  
) where dept = 'eecs';
```

```
select avg(sal)  
from emp  
group by dept  
having dept = 'eecs';
```

• Not always possible; consider

```
create view sals as {  
    select distinct dept, sal  
    from emp  
}
select avg(sal) from sals
```

```
select avg(sal) from (  
    select distinct dept, sal  
    from emp  
)
select avg(distinct sal)  
from emp
```
Flatten this query (*departments where number of machines is more than number of employees*):

```sql
SELECT dept.name
FROM dept
WHERE dept.num_machines >=
     (SELECT COUNT(emp.*)
      FROM emp
      WHERE dept.name=emp.dept_name)
```

What happens if there is a department with no employees?
SELECT dept.name
FROM dept
LEFT OUTER JOIN emp ON (dept.name=emp.dept_name)
GROUP BY dept.name,
        dept.num_machines
HAVING dept.num_machines >= COUNT(emp.*)

SELECT dept.name
FROM dept
LEFT OUTER JOIN emp ON (dept.name=emp.dept_name)
GROUP BY dept.name,
        dept.num_machines
HAVING dept.num_machines >= COUNT(emp.*)
Query Processing Steps

• Admission Control
• Query Rewriting
• Plan Formulation (SQL $\rightarrow$ Tree)
• Optimization
Plan Formulation

emp (eno, ename, sal, dno)
depth (dno, dname, bldg)
kids (kno, eno, kname, bday)

SELECT ename, count(*)
FROM emp, dept, kids
AND emp.dno=dept.dno
AND kids.eno=emp.eno
AND emp.sal > 50000
AND dept.name = 'eecs'
GROUP BY ename
HAVING count(*) > 7
Query Optimization

Logical planning:
operator ordering
(exponential search space)

Physical planning:
operator implementation
& access methods
(indexes vs heap files)
Joins and Ordering

• Consider a nested loop join operator of tables Outer and Inner

  for tuple1 in Outer
    for tuple2 in Inner
      if predicate(tuple1, tuple2) then
        emit join(tuple1, tuple2)

• What if Inner is itself a join result?
• Plans might be “left-deep” or “bushy”
Plan for Next Few Lectures

Query System
- Parser
- Rewriter
- Planner
- Executor

Storage System
- Access Methods
- Buffer Manager
- Lock Manager
- Log Manager

Today
+Lec 6

Lec 7

Lec 8 – Join Algos

Optimizer (Lec 10)
Query Execution

• Executing a query involves chaining together a series of operators that implement the query.

• Operator types:
  - **scan** from disk/mem
  - **filter** records
  - **join** records
  - **aggregate** records

\[ \sigma_{\text{birthday} \ldots} \]
\[ \pi_{\text{movieTitle}} \]
\[ \bowtie \text{starName = name} \]
\[ \pi_{\text{starsIn}} \]
\[ \text{movieStar} \]
Physical Layout

• Arrangement of records on disk / in memory
• Disk / memory are linear, tables are 2D
  – "Row Major" - Row at a time
Physical Layout

• Arrangement of records on disk / in memory
• Disk / memory are linear, tables are 2D
  – ”Row Major” - Row at a time
  – “Column Major” Column at a time

For now, let’s assume row-major!
Accessing Data

• Access Method: way to read data from disk

• Heap File: unordered arrangement of records
  – Arranged in pages

Header: Start offset of each record, which parts of page are occupied, etc
Heap Scan

- Read Heap File In Stored Order
  - Even with a predicate, read all records

<table>
<thead>
<tr>
<th>Header</th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>R4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Page 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Header</th>
<th>R4</th>
<th>R5</th>
<th>R6</th>
<th>R7</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Page 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Header</th>
<th>R8</th>
<th>R9</th>
<th>R10</th>
<th>R11</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Page 3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

...
Index

• Index maps from a value or range of values of some attribute to records with that value or values

• Several types of indexes, including trees (most commonly B+Trees) and hash indexes

API:
**Lookup**(value) → records
**Lookup**(v1 .. vn) → records

Value is an attribute of the table, called the “key” of the index
Tree Index

Index File

Heap File

Attr1

::

Attrn

Hdr

R

1

R

2

R

3

R

4

R

5

R

6

R

7

R

8

R

9

R

10

R

11

<3

≥3, <5

≥5, <7

≥8, 9

0

1

2

2

2

3

4

5

6

8

9

9

3

2

9

4

6

1

0

2

9

8

2

5
Index Scan

Traverse the records in Attr1 order, or lookup a range

Attr1 >= 6
&
Attr1 < 9

Note random vs sequential access!
Clustered Index

- Order pages on disk in index order

Attr1: <3, ≥3, <5, ≥5, <7, ≥8, <9

Index File

Heap File
Clustered Index

- Order pages on disk in index order

Eliminates random I/O for index scans on Attr1 (but only Attr1!)

Index File

Heap File
Let’s take a short break
Connecting Operators: Iterator Model

Each operator implements a simple iterator interface:

- `open(params)`
- `getNext() → record`

Any iterator can compose with any other iterator

\[
\begin{align*}
&\text{it1 = Scan.open(“movieStar”, …)} \\
&\text{it2 = Filter.open(it1, bday=x, …)} \\
&\text{it3 = Scan.open(“starsIn”, …)} \\
&\text{it4 = Join.open(it2, it3, starName=name)} \\
&\text{it5 = Proj.open(it4, movieTitle)}
\end{align*}
\]
Iterator Model

\[
\begin{align*}
\text{it1} &= \text{Scan.open}(\text{“movieStar”, …}) \\
\text{it2} &= \text{Filter.open}(\text{it1}, \text{bday}=x, \text{…}) \\
\text{it3} &= \text{Scan.open}(\text{“starsIn”, …}) \\
\text{it4} &= \text{Join.open}(\text{it2}, \text{it3}, \\
&\quad \text{starName}=\text{name}) \\
\text{it5} &= \text{Proj.open}(\text{it4}, \text{movieTitle})
\end{align*}
\]
What is SimpleDB?

- A basic database system
- SQL Front-end (Provided for later labs)
  - Heap files (Lab 1)
  - Buffer Pool (Labs 1-6)
  - Basic Operators (Labs 1 & 2)
    - Scan, Filter, JOIN, Aggregate
  - Query optimizer (Lab 3)
  - Transactions (Lab 4)
  - B-Tree Indexes (Lab 5)
  - Recovery (Lab 6)
- Javadoc is your friend!
Database

Singleton Database:

- Database.getCatalog():
  - Catalog
  - => List of DB tables

- Database.getBufferPool():
  - BufferPool
  - => Caches DB pages in memory

- Database.getLogFile():
  - LogFile
  - (Ignore for Lab 1)
Catalog:

<table>
<thead>
<tr>
<th>DbFile ID</th>
<th>Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>001</td>
<td>Table1</td>
</tr>
<tr>
<td>002</td>
<td>Table2</td>
</tr>
<tr>
<td>003</td>
<td>Table3</td>
</tr>
</tbody>
</table>

Table:

- DbFile file
- String name
- String primary_key

=> Stores a list of all tables in the database
Buffer Pool

Buffer Pool:

<table>
<thead>
<tr>
<th>Page ID</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>001</td>
<td>Page1</td>
</tr>
<tr>
<td>003</td>
<td>Page3</td>
</tr>
<tr>
<td>007</td>
<td>Page7</td>
</tr>
</tbody>
</table>

Page:
Pageld id
Tuple tuples[]
Byte header[]

=> Caches recently accessed database pages in memory
HeapPage
(Implements Page)

Heap Page:
- HeapPageId pid
- TupleDesc td
- Byte header[]
- Tuple tuples[]

Tuple Descriptor:
- Field1 Type
- Field1 Name
- Field2 Type
- Field2 Name
- Field3 Type
- Field3 Name
- ...

Header:
- 01100110
- 11111111
- 11101101
- ...

Tuples:
- Empty
- Tuple1
- Tuple2
- ...

Tuple:
- Field1
- Field2
- Field3
- ...

*Fields and Tuples are Fixed Width!*
HeapFile
(Implements DbFile)

File (on disk):

Heap File:
- File file
- TupleDesc td
- DbFileIterator it

Tuple Descriptor:
- Field1 Type
- Field1 Name
- Field2 Type
- Field2 Name
- Field3 Type
- Field3 Name

Iterate through Tuples in Heap Pages:
- Page1
- Page2
- Page3
- ...

-
SeqScan
(Implements DbIterator)

- DbIterator class implemented by all operators
  - open()
  - close()
  - getTupleDesc()
  - hasNext()
  - next()
  - rewind()

- Iterator model: chain iterators together
  - Use DbFileIterator from HeapFile
```java
// construct a 3-column table schema
Type types[] = new Type[]{ Type.INT_TYPE, Type.INT_TYPE, Type.INT_TYPE};
String names[] = new String[] { "field0", "field1", "field2"};
TupleDesc descriptor = new TupleDesc(types, names);

// create the table, associate it with some_data_file.dat
// and tell the catalog about the schema of this table.
HeapFile table1 = new HeapFile(new File("some_data_file.dat"), descriptor);
Database.getCatalog().addTable(table1);

// construct the query: we use a simple SeqScan, which spoonfeeds
// tuples via its iterator.
TransactionId tid = new TransactionId();
SeqScan f = new SeqScan(tid, table1.id());

// and run it
f.open();
while (f.hasNext()) {
    Tuple tup = f.next();
    System.out.println(tup);
}

f.close();
Database.getBufferPool().transactionComplete();```
HeapFileEncoder.java

- Because you haven’t implemented insertTuple, you have no way to create data files

- HeapFileEncoder converts CSV files to HeapFiles

- Usage:
  - `java -jar dist/simplesdb.jar convert csv-file.txt numFields`

- Produces a file `csv-file.dat`, that can be passed to HeapFile constructor.