What happens inside?

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

SELECT name, addr, balance
FROM accounts
WHERE userid = ?
What happens inside?

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

```
SELECT image, imgid
FROM planet_images
WHERE likelyPlanet(image)=TRUE
```
What happens inside?

Admission Control

Dispatch and Scheduling

Process Manager (Section 2)

Client Communications Manager

Local Client Protocols

Remote Client Protocols

Catalog Manager

Memory Manager

Administration, Monitoring & Utilities

Replication and Loading Services

Batch Utilities

Shared Components and Utilities (Section 7)

Query Parsing and Authorization

Query Rewrite

Query Optimizer

Plan Executor

Relational Query Processor (Section 4)

Access Methods

Buffer Manager

Lock Manager

Log Manager

Transaction Manager (Section 5 & 6)

DDL and Utility Processing

Transaction Manager (Section 6)
What happens inside?

Why do you need admission control?
Why is scheduling important?
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

Local Client Protocols
Remote Client Protocols
Client Communications Manager

Query Parsing and Authorization
Query Rewrite
Query Optimizer
Plan Executor
Relational Query Processor (Section 4)

Transaction Manager (Section 2)

Access Methods
Buffer Manager
Lock Manager
Log Manager

Transaction Storage Manager (Sections 5 & 6)

Catalog Manager
Memory Manager
Administration, Monitoring & Utilities
Replication and Loading Services
Batch Utilities
Shared Components and Utilities (Section 7)
What happens inside?

Admission Control

Local Client Protocols
Remote Client Protocols

Client Communications Manager

Catalog Manager

Memory Manager

Administration, Monitoring & Utilities

Replication and Loading Services

Batch Utilities

Shared Components and Utilities (Section 7)

Process Manager (Section 2)

Query Parsing and Authorization

Query Rewrite

Query Optimizer

DDL and Utility Processing

Plan Executor

Relational Query Processor (Section 4)

Access Methods

Buffer Manager

Lock Manager

Log Manager

Transactional Storage Manager (Sections 5 & 6)
What happens inside?

- Admission Control
- Dispatch and Scheduling
- Process Manager (Section 2)
- Local Client Protocols
- Remote Client Protocols
- Catalog Manager
- Memory Manager
- Administration, Monitoring & Utilities
- Replication and Loading Services
- Batch Utilities
- Shared Components and Utilities (Section 7)

- Query Parsing and Authorization
- Query Rewrite
- Query Optimizer
- DDL and Utility Processing
- Relational Query Processor (Section 4)

- Access Methods
- Buffer Manager
- Lock Manager
- Log Manager
- Transactional Storage Manager (Sections 5 & 6)
What happens inside?

Admission Control

Client Communications Manager

Local Client Protocols

Remote Client Protocols

Query Parsing and Authorization

Query Rewrite

Query Optimizer

DDL and Utility Processing

Plan Executor

Relational Query Processor (Section 4)

Access Methods

Buffer Manager

Lock Manager

Log Manager

Transactional Storage Manager (Sections 5 & 6)

Catalog Manager

Memory Manager

Administration, Monitoring & Utilities

Replication and Loading Services

Batch Utilities

Shared Components and Utilities (Section 7)
DB Core Components

Admission Control

Connection Management

Query System

- Parser
- Rewriter
- Planner
- Executor

Optimizer

Storage System

- Access Methods
- Buffer Manager
- Lock Manager
- Log Manager
Flow of a Query

Query System
- Parser
- Rewriter
- Planner
- Executor

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

Admission Control
Connection Management

Query System
- SQL
  - Parse Tree
  - Optimizer
    - Query Plan

Storage System

Select
FROM StarsIn, MovieStar
WHERE starName = name
AND birthday LIKE "%'1960"

Project
movieTitle

Select
FROM movieStar
WHERE starName = name
AND birthday...

Project
movieTitle

Star starName = name

movieStar

StarsIn

movieTitle

π

MovieStar

StarsIn

movieTitle

π

Access Methods

Buffer Manager

Lock Manager

Log Manager
Plan for Next Few Lectures

Admission Control

Connection Management

Query System

Parser

Rewriter

Planner

Executor

Storage System

Access Methods

Buffer Manager

Lock Manager

Log Manager

Today

+Lec 5

Today

Lec 6

Lec 7 – Join Algos

Optimizer (Lec 9)
Query Processing Steps

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

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

• View Substitution
• Predicate Transforms
• Subquery Flattening
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';
Predicate Transforms

• Remove & simplify expressions, improve

• Constant Simplification
  WHERE sal > 1000 + 4000 \(\Rightarrow\) WHERE sal > 5000

• Exploit constraints
  a.did = 10 and a.did = dept.dno and dept.dno = 10

• Remove redundant expressions
  a.sal > 10k and sal > 20k
Predicate Transforms

• Remove & simplify expressions, improve

• Constant Simplification

  WHERE sal > 1000 + 4000 → WHERE sal > 5000

• Exploit constraints

  a.did = 10 and a.did = dept.dno and dept.dno = 10

• Remove redundant expressions

  a.sal > 10k and 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 = 'eeecs';
```

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

Can you come up with an example where this is not possible?
Subquery Flattening

• Many Subqueries Can Be Eliminated

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

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

• Not always possible; consider

```sql
create view sals as { 
    select distinct dept, sal 
    from emp 
} 
```

```sql
select avg(sal) from sals
```

```sql
select avg(sal) from ( 
    select distinct dept, sal 
    from emp 
) 
```

```sql
select avg(distinct sal) from emp
```
Study Break (Tricky)

Flatten this query (departments where number of machines is more than number of employees):

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

A

SELECT dept.name
FROM dept d, emp e
WHERE d.name=e.dept_name
    and d.num_machines >= COUNT(e.*)
GROUP BY d.name, d.num_machines

B

SELECT dept.name
FROM dept d, emp e
WHERE d.name=e.dept_name
GROUP BY d.name
HAVING COUNT(d.num_machines) >= COUNT(e.*)

C

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

D
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

\[
\begin{align*}
\text{emp} & (\text{eno}, \text{ename}, \text{sal}, \text{dno}) \\
\text{dept} & (\text{dno}, \text{dname}, \text{bldg}) \\
\text{kids} & (\text{kno}, \text{eno}, \text{kname}, \text{bday}) \\
\end{align*}
\]

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

Logical planning: operator ordering (exponential search space)

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

Storage model & access methods?
Join 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

Admission Control

Connection Management

Query System

Parser

Rewriter

Planner

Executor

Optimizer (Lec 9)

Lec 7 – Join Algos

Storage System

Access Methods

Buffer Manager

Lock Manager

Log Manager

Today

Today + Lec 5

Lec 5

Lec 6

Lec 9
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

→ Requires a model of data representation
Physical Layout

• Arrangement of records on disk / in memory
• Disk / memory are linear, tables are 2D

How would you store the table on disk?

Knowing that you must efficiently support inserts, deletes, and that some records are more often read than others?
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!
How would you store records on disk?
Accessing Data

- **Access Method**: way to read data from disk
- **Heap File**: unordered arrangement of records
  - Arranged in pages
  - You read/write/cache data in the granularity of pages.

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

```
Get Page 3 = Page# * PageSize
```
Heap Scan

- Read Heap File In Stored Order
  - Even with a predicate, read all records
Hardware (e.g., SSDs) and OS (e.g., virtual memory) also use pages. They often are 4KB large.

Why does a database management introduce yet another paging mechanism?
Page designs

Strawman idea: Keep track of tuples in a page?

Any problems with this design?
Page designs

Strawman idea: Keep track of tuples in a page?

- What happens with deletes?
- What happens with variable length tuples (e.g., variable length strings)?
Slotted pages

Common layout scheme

• Slot array maps "slots" to tuples starting position
• The header keeps track of:
  → The # of used slots
  → The offset of the starting location of the last slot used.
Slotted pages

How would you simplify the layout if tuples have a fixed length?

Do you need to store the slot map?
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

<3  ≥3, <5  ≥5, <7  ≥8, 9

0 1 2 2 2

3 4 5 6

8 9 9

Hdr R1 R2 R3 R4

3 2 9 4

Hdr R4 R5 R6 R7

6 1 0 2

Hdr R8 R9 R1 R1

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
Clustered Index

- Order pages on disk in index order

Eliminates random I/O for index scans on Attr1 (but only Attr1)!
Let’s take a short break
Connecting Operators: Iterator Model

Each operator implements a simple iterator interface:

- `open(params)`
- `getNext()` \rightarrow record
- `close()`

Any iterator can compose with any other iterator

```
It1 = Scan.open("movieStar", ...)
It2 = Filter.open(it1, bday=x, ...)
It3 = Scan.open("starsIn", ...)
It4 = Join.open(it2, it3, starName=name)
It5 = Proj.open(it4, movieTitle)
```
Iterator Model

it1 = Scan.open("movieStar", ...)  

it2 = Filter.open(it1, bday=x, ...)  

it3 = Scan.open("starsIn", ...)  

it4 = Join.open(it2, it3, starName=name)  

it5 = Proj.open(it4, movieTitle)
Lab1: What is GoDB?

A basic database system implemented in Go

• A simple storage layer, based on Heap Files (Lab 1)

• A buffer pool for caching pages and implementation page-level locking for transactions (Labs 1-3)

• A set of operators (Labs 1 & 2): Scan, Filter, Join, Aggregate, Order By, Project ...

• A SQL parser (Lab 2), which we implement for you

• Simple transactions (Lab 3)

• Previous years we included recovery, B+Trees, and query optimization, but have reduced the labs because this is our first year in Go.
  – Students in 6.5831 may implement one of these for their final project
What is GoDB Missing?

- Focus is on a simple architecture rather than a complete or high-performance implementation
- Only supports fixed length records with strings and ints
- Only supports sequential scan access methods
- No NULLs
- Uses a simple iterator method, so not super efficient
GoDB Storage Layout

- Each table is stored in one file on disk, called a *heap file*
  - Heap files are an unordered collections of records
  - Only way to access records from a heap file is to scan from beginning to end: “Sequential scan” via an iterator
- Each heap file consists of a number of fixed size heap pages
- Each heap page contains a number of fixed size tuples
- Methods in `heap_file.go` are used to access the contents of the heap file
Tuples and Tuple Descriptors

• In a given heap file, each tuple has the same layout
• Layout is specified by a TupleDesc object, which specifies the field names and types in the tuple

// FieldType is the type of a field in a tuple, e.g., its name, table, and [godb.DBType]. // TableQualifier may or may not be an empty string, depending on whether the table // was specified in the query
type FieldType struct {
    Fname string
    TableQualifier string
    Ftype DBType
}

// TupleDesc is "type" of the tuple, e.g., the field names and types
type TupleDesc struct {
    Fields []FieldType
}
Tuples and Tuple Descriptors (cont.)

- Tuple objects contain the values of each record in Fields
- Field is an interface, implemented by IntField and StringField
- All ints are 64 bits; all string are StringLength characters, padded with zeros

```go
// Tuple represents the contents of a tuple read from a database
// It includes the tuple descriptor, and the value of the fields
type Tuple struct {
    Desc  TupleDesc
    Fields []DBValue
    Rid   recordID // used to track the page and position this page was read from
}
```
HeapFile (table1)

Note: you need a way to deal with deletes!
Buffer Pool

• Buffer pool is an in-memory cache of pages
• Allows GoDB to control how much memory is used and support tables larger than memory
• For transactions, will be responsible for implementing page-level locking and two-phase commit (not until lab 3)

• All iterators and operators should use the buffer pool GetPage method to access pages from heap files
• Only the heap file readPage method directly reads data from disk
Iterators

- Each database operator (filter, project, join, etc) implements an *Iterator*

```go
type Operator interface {
    Descriptor() *TupleDesc
    Iterator(tid TransactionID) (func() (*Tuple, error), error)
}
```

- Iterator() returns a function that iterates through the operator’s records
- Most operators take a child operator as a part of their constructor

```go
func NewIntFilter(constExpr Expr, op BoolOp, field Expr, child Operator) (*Filter[int64], error) { ... }
```

- Heap file Iterator iterates through pages on disk; other operators iterate through their child tuples
  - E.g., filter iterates through child tuples, applied the filter to them, and returns satisfying tuples
Iterator Implementation

- Returns a function that when called returns the next tuple
- Needs to keep state of where it was in its child

```go
func (f *Filter[T]) Iterator(tid TransactionID) (func() (*Tuple, error), error) {
    childIter, _ := f.child.Iterator(tid) // childIter is current iterator state
    ...
    return func() (*Tuple, error) {
        for {
            // get child tuple from childIter
            // get tuple fields (e.g., using EvalExpr)
            // apply predicate
            // if matches, return tuple
            // else go onto next tuple
            }, _
        }
    }
```
Example

Client \( \xrightarrow{\text{Iterator()}} \) Project(hf)

\( f() \)

hf: HeapFile

bp: BufferPool

pageCache
Example

Diagram:
- Client
- Project(hf)
  - Iterator()
  - f()
  - f2()
- hf:HeapFile
- bp:BufferPool
- pageCache
Example
Example
Deleting Records and Rids

• Consider a query like:
  DELETE FROM x WHERE f > 10
This is translated into a plan like

Q: How does the delete operator know which records to delete?
A: Each record from the HeapFile is annotated with a record id that is used to identify the position of the record in the heap file to be deleted
Deleting Records and Rids

// Remove the provided tuple from the HeapFile. This method should use the
// [Tuple.Rid] field of t to determine which tuple to remove.
// This method is only called with tuples that are read from storage via the
// [Iterator] method, so you can so you can supply the value of the Rid
// for tuples as they are read via [Iterator]. Note that Rid is an empty interface,
// so you can supply any object you wish. You will likely want to identify the
// heap page and slot within the page that the tuple came from.
func (f *HeapFile) deleteTuple(t *Tuple, tid TransactionID) error {

  • deleteTuple will be called by the delete operator
  • Using the t.Rid object, you can clear out the position in the heap file containing
    the record
  • Your heapfile implementation supplies the Rid in the iterator, and so you can
    identify this position however you like
  • A standard Rid implementation is a page number and a slot within the page
    • Recall that all pages have the same number of slots
func computeFieldSum(fileName string, td TupleDesc, sumField string) (int, error) {

    //Create buffer pool
    bp := NewBufferPool(10)

    hf, err := NewHeapFile("myfile.dat", &td, bp)
    ...
    err = hf.LoadFromCSV(CSVfile, true, ",", false)

    //find the column
    fieldNo, err := findFieldInTd(FieldType{sumField, "", IntType}, &td)

    //Start a transaction -> we will do the implementation in another lab
    tid := NewTID()
    bp.BeginTransaction(tid)
    iter, err := hf.Iterator(tid)

    //Iterate through the tuples and sum them up.
    sum := 0
    for {
        tup, err := iter()
        f := tup.Fields[fieldNo].(IntField)
        sum += int(f.Value)
    }

    bp.CommitTransaction() //commit transaction
    return sum, nil //return the value
Have Fun!

- Start early
- Let us know what you find confusing on Piazza!