**Embedded SQL**
At compile time: Static SQL • SQL is hard-coded into the application, with limited parameters replaceable with variable values at run time • SQL is analyzed and compiled at preprocessor, compile time, so run time is more efficient
At run time: Dynamic SQL • SQL is generated on-the-fly during application execution • SQL is dynamically analyzed and compiled by the DMBS during run time
2 tier - Client / Server Client->DB server (network)
3 tier, web, web service Client-(public network)->Web server App servers->DB server
**Components [5]**: - Network connectivity - DB access library - DB call interface library - Connection - Query Execution - Results
**Cursors** • SQL queries return bags of records, with no a priori bound on the number of columns or records. No such data structure exists in most languages. • In addition, applications may dynamically discover the schema of the result set, or request records to be returned in chunks • SQL supports a mechanism called a cursor to handle this.
– Can declare a cursor on a table or query statement (which generates a table).
– Can open a cursor, and fetch all rows or fetch rows one at a time, until all rows have been retrieved.
– Some cursor implementations are synchronous, some async • May also be possible to modify/delete row pointed to by a cursor.
– Cursor can report conditions (e.g., end of rows)

## Dynamic SQL Application Flow

- Open a connection to the database – Requires an account and credentials to use for authentication – Needs the db server name and network connectivity – $connection = pg_connect("host=$hostname dbname=$databasename user=$username password=$password")
- Form a SQL statement, store in a string variable – Whatever sw programming logic is applicable – UI, others – $query = "SELECT …. FROM …. WHERE …."
- Execute the SQL statement on the db and get the results – $result = pg_query($connection, $query)
- Consume the results of the query using a CURSOR – pg_num_rows($result) – $row = pg_fetch_row($result) – echo "First Name: $row[0] \n"; – Or copy into your applications datastructures
- Close the connection to your database – pg_close($connection);

**Java** - Exposed through libraries called JDBC (Java DataBase Connectivity • Using package java.sql.* • All of the principles of cursors still apply – They are now encapsulated in object methods • A Java Cursor is called a **ResultSet Object** • Column names and positions are stored in a ResultSetMetaData object
**.NET** - Exposed through the libraries System.Data.* – We'll focus on System.Data.OleDb • Can be used in the Microsoft .NET framework (Basic, C#, Managed C++, JScript) – We'll use C# • Object-Oriented
A .NET Cursor is a **OleDbDataReader Object.**

Stored Procedures Can create procedural routines that execute on the server to move some of the compute from client to the db We won't get into details of writing these in this class, but it's common to put complex sql interactions into stored procedures • to minimize network – chunky, not chatty • For performance - stored procedures are compiled, execution is faster than equivalent dynamic sql • Security – protects against SQL injection • Separation of concerns • Isolation • Maintenance
Why wouldn't you use them? • If parameterizing your possible queries makes them much more complex • If you need the flexibility of dynamic SQL

**Memory Hierarchy** – speed (TPT, Latency), Cost, Capacity
[Storage: Block level IO, Very Slow] – [Memory: Byte level IO, Fast]
No matter the technology, there will be smaller, faster, more expensive storage and a larger, slower, cheaper storage
Components of a Disk - platters are always spinning (say, 5400rps). • one head reads/writes at any one time.
Each track is made up of fixed size sectors. All the tracks that you can reach from one position of the arm is called a cylinder. Block size is a multiple of sector size.
To read a record: • position arm (seek) • engage head • wait for data to spin by • read (transfer data)
Time to access (read/write) a disk block: – seek time (moving arms to position disk head on track) – rotational delay (waiting for block to rotate under head) – transfer time (actually moving data to/from disk surface) Disk access time = seek time + rotational latency + transfer time
Key to lower I/O cost: reduce seek/rotation delays! (to reduce transfer time, you need multiple disks or multiple arms)
How to minimize the cost of Disk I/Os – Consecutive pages on same track, followed by
– Consecutive tracks on same cylinder, followed by – Consecutive cylinders adjacent to each other – First two incur no seek time or rotational delay, seek for third is only one track.

DBs store data in files • Most common organization is row-wise storage • On disk, a file is split into blocks • Each block contains a set of tuples • In the example, we have 4 blocks with 2 tuples each • File can be sorted (sequential file) or unsorted (heap)
Files and Pages • Disk space is divided into large files • Files are divided into pages, usually 8KB/page • Disk I/O operations are performed at the page level. The dbms reads or writes whole data pages. • DBMSs have specific methods of managing – Which pages belong to which objects – Which pages are free – Identifying a specific record in a specific page for a specific object

**Indexes - (**Guide to Pioneer Courthouse Square)
An Index is an additional file & data structure that speeds up selections on the search key field(s) • An index transforms a search key k into a data entry k*. • Given k*, you can get to the record(s) that have the search key k in one I/O
You can have many indices for one table – Every index data structure itself is stored in order of its search keys – If the records of the underlying table are stored in order of an Index, then that index is called a Clustering Index on the Table – There can only be one Clustered Index on a Table
A table's primary key is a unique, non-null constraint • Many dbms (Postgres) creates tables indices for every unique constraint
A table's clustered Index determines the order that the table's rows will be stored on disk. • It is common to make the primary key index a clustering index for the table, but it's not required • Some dbms's set the primary key to a clustering index by default
Tree-structured indexes support range searches and equality searches.
B+ tree: dynamic – index is adjusted as records are inserted and deleted in the file. Index remains [balanced]
Non Leaf nodes - Only Contain index and pts. Leaf nodes - Contain index and data entries - Are chained
k* we can store: 1. Data record having key value k, or 2. RID of data record having search key value k. 3. list of RIDs of records having search key k. (*RID = Record id: location in file structure. (X:Y:Z), points to data file X, page Y and slot Z)
Hash-Based Indexes • Only good for equality search(selections). • Index is a collection of buckets. – Bucket = primary page plus zero or more overflow pages. – Buckets contain data entries. Hashing function h: h(r) = bucket in which (data entry for) record r belongs. h looks at the search key fields of r. – No need for “index entries” in this scheme.
VS(1)Hash indexes can only be used for equality searches – Search for records with City = 'Paris' – Hash 'Paris', find the exact bucket containing Paris records – O(1) for Hash, vs O(Log(n)) for B+ tree • (2)Hash’s can't be used for range queries – Search for records with City > 'P' and City < 'S' – Any number of hash buckets, in no particular order. You need to know the possible city values in that range, and hash each one • Basically turning a range into a set of equality conditions (3) Buckets can get filled up and need to be split, rehashed
VS index sizesTherefore direct access costs 2-3 disk probes, not much more than hashing.
Costs of an Index • If you define an index in your database, you will incur three costs – Space to store the index – Updates to the search key will be slower – The optimizer will take longer because it has more choices • There is one advantage to having an index – Some queries run faster (better be sure about this) – Including looking for duplicate primary key
Clustered Index: Records are organized based on search key for the index
(Dense): [index 1 ->data1]  (Sparse): [index 1 -> data 1,2,3]
Unclustered Index: Records NOT Sorted on Search Key
Relation: Clustered, sparse indexes are smaller; they work well for range searches and sorting. But…some useful optimizations are based on dense indexes. Note: one file can have at most one single-attribute clustered index - all of the additional single-attribute indices must be unclustered. Sparse-clustered. Dense-non & clusted.
Multi-column Indexes
Practical Indexing

**Join Algorithms**
Simple Nested Loop Join – We read every PAGE in S for every ROW in R. Cost: M + (pR * M) * N
Page-oriented Nested Loops Join – read every PAGE in S for every PAGE in R. Cost: M + M*N (for R) 都反(s)
Multi-Page, Block oriented Nested Loops Join – We read every PAGE in S for every GROUP of pages in R Cost: M + (M/(B-2))*N
Index Nested Loops Join – We read 2-4 PAGES in S for every ROW in R. Cost: M + (M*pR ) * (2-4)
Sort Merge Join – Sort R on join attribute.Sort S on join attribute.Merge R and S [best] Cost to sort R + Cost to sort S + (M+N) .[Worst] Cost to sort R + Cost to sort S + (M*N)
If both R and S can be sorted in 2 passes. Cost is: 4M+4N+(M+N) = 5*(M+N)
Optimized*Merge one pass of sorting R and S, 2M+2N+(M+N) = 3*(M+N)
Hash Join – We partition S, R into Hash buckets, and merge bucket by bucket
- Cost to hash-partition R: 2M • Cost to hash-partition S: 2N • Cost to join partitions: M+N = 3*(M+N)

**Query Execution Planning**

**Transaction** (Needs to be done “all or nothing”) is: – one “complete” set of actions – defined by the user (meaningful to the application) – establishes where certain integrity constraints are enforced.

**For concurrency control purposes (inside DBMS):**

**– a transaction is one atomic unit of work. Thus we must be able to undo work-so-far if incomplete**

**– DBMS cares only about the reads/writes to the DB**

**– DBMS views a transaction as (only) a sequence of reads, writes plus commit & rollback (abort)**

**User (SQL or application developer) must indicate:**

– BEGIN TRANSACTION;

– [SQL statements] • read/write/modify statements intermixed with other programming language statements

plus either – COMMIT; - indicates successful completion

or – ROLLBACK; - indicates program wants to roll back (abort) the transaction

In postgres, phpPgAdmin, you'll also want to explicitly set your isolation level after beginning a txn, using: - SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;

Higher throughput, lower response times.

**ACID**: Atomicity-**recovery system** Consistency-**DB design** Isolation-**concurrency** Control system durability-**recovery system**

**Isolation** – during Transaction 1's execution, the data it sees is as-if no other transactions are operating on that data

- If Transaction 1 completes, and then Transaction 2 changes the results of transaction 1… still isolated. – As long as T1 and T2 started with consistent data, and ended with consistent data (according to itself), committed, and ended, they're isolated (during their execution)
- If Transaction 1's performance is affected by the load on the system … still isolated – Isolation refers to data integrity isolation, not performance isolation
- If Transaction 1 aborts and rolls back because it can't obtain required read/write locks … still isolated – If we can't enable Transaction 1 to complete (in some time) in a data isolated way, it's ok to abort it and roll back.

“READ UNCOMMITTED” – allows [dirty reads], unrepeatable reads, and “phantoms” “READ COMMITTED” – allows unrepeatable reads and phantoms “REPEATABLE READ” – allows phantom rows “SERIALIZABLE” – full isolation. Phantom: A record that appears or disappears during a transaction. Unrepeatable Read: may read the same data twice and get two different answers.

**Schedule (ensure isolated):** An interleaving of actions from a set of transactions, where the actions of each individual transaction are in the original order.

– Represents an actual sequence of database actions.

– Example: R1 (A), W1 (A), R2 (B), W2 (B), R1 (C), W1 (C)

– In a complete schedule, each transaction ends in commit or abort.

- Initial State of DB + Schedule => Final State of DB

**Serializable Schedule => Isolated Transactions**

- Serial schedules: – Run transactions one at a time, in a series. (Different orders might give different results.)
- Serializable schedules: – Final state must be the same final state as produced by one of the serial schedules. – Must appear to each transaction as if the transactions that precede it ran sequentially

**Create a precedence graph**:

• A node for each transaction Ti

• An edge from Ti to Tj whenever an action in Ti conflicts with, and comes before an action in Tj

**The schedule is conflict-serializable iff the precedence graph is acyclic**

The schedule is non-conflict-serializable if the precedence graph is cycle

**Transaction must get a Lock** – before it can read or update a data item • There are two kinds of locks: 共享锁shared (S) locks and 排他锁exclusive (X) locks

- To read a record you MUST get an S (or X) lock To modify or delete a record you MUST get an X lock • Lock info maintained by a “lock manager”

**Strict 2PL guarantees (conflict) serializability**

- Can prove that a Strict 2PL schedule is equivalent to the serial schedule in which each transaction runs instantaneously at the time that it commits
- This is huge: A property of each transaction (2PL) implies a property of any set of transactions (serializability) No need to check serializability of specific schedules
- Most DBMSs use 2PL to enforce serializability
- Equivalent serial order is order of commits

**Recovery**

Goal: Design a system such that after a system failure, we can return to a consistent, valid state.

The state of data when the system resumes reflects • All data changes from all transactions that committed prior to system failure • Restoration of initial state for all transactions that rolled back prior to system failure • No data changes from uncommitted transactions running at the time of the system failure

**System failure**

1)Many possibilities, but consider that we pull the plug on the whole system with no notice, at any point in the flow of a txn 2)Could be system failure, dbms crash, OS crash, network crash, disk crash, lightning, spilled beer…. 3)Approach: We will look at logging changes, which supports recovery from application and system failures • Will write log records to a transaction log that tracks changes that each transaction makes

**Use a Separate Log of Transaction Changes**

- During a transaction, change an item in memory
- But also write information to a separate log file on disk about that change (item, old value, etc)
- Always append to the log file, so there's no contention or locking on any "record" in the log file
- **HOW?** Initial design was to write the previous values to the log file, so we could "undo" them if needed

**Some terminology**

When moving from the Simple Model to a Logging Model, sometimes data in memory doesn't match data on disk

- Dirty page: changed in memory, not on disk
- memory-resident data can be lost in a crash.

Log records capture changes that transactions make; and are used to "repair" data on disk after a crash.

Log records are written to disk along with data changes

**Undo logging**: Write the old values of changed data items, undo changes of uncommitted transactions on recovery log record precedes changes to disk precedes commit (supports undo'ing changes to disk if failure before commit) (forces pages to be written at txn commit) we have to delay commit of a transaction until all its updates go to disk. Undo written but uncommitted changes to disk

**Redo logging:** Write the new values of changed data items, redo changes of committed transaction on recovery log record precedes commit precedes changes to disk (supports redoing changes to disk if failure after commit but before disk writing.) (delays writing pages until all txns touching that page commit) we can't release a dirty page from memory if the transaction hasn't reached commit. Redo unwritten but committed changes to disk

**Transaction Commit** - Push all log records for this transaction to disk.

**Benefit** Do Undo/Redo logging: Log records hold both old and new values for changes – Don't have to write out all data before committing – Can write uncommitted data to disk -With this, pages in memory can be written to disk at any time, independent of whether or not they hold dirty data — Can use log files to create multiple copies of a live database! • Log File Shipping• There are other technologies for creating copies of a db, including log file replication and db mirroring.

## Simple Nested Loops Join (very naive)

M pages in R
N pages in S
$p_R$ , $p_S$ rows per page

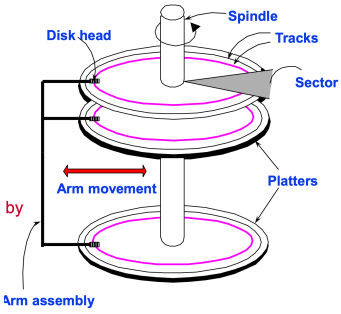
```
Join on ith column of R and jth column of S
foreach row r in R do
        foreach row s in S do
                if ri == sj then add <r, s> to result
```

## Page-oriented Nested Loops Join

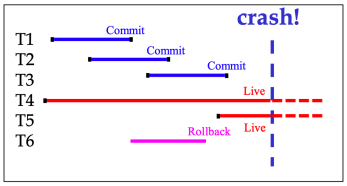
```
for each page of rows r in R do
        for each page of rows s in S do
                (match all combinations in memory)
                if ri == sj then add <r, s> to result
```

## Index Nested Loops Join

```
foreach row r in R do
        foreach row s in S where ri == sj do
                add <r, s> to result
```




## Which Transactions are Recovered and How?




- Desired Behavior after system restarts:
  - T1, T2 & T3 committed and must be durable
    - Ensure durability by redoing each Txn using the log.
  - T4 & T5 were still running and must be atomic.
    - Ensure this by undoing the Txn using the log.
  - T6 was rolled back before the crash and must leave no updates
    - Ensure this by undoing the Txn using the log.

## Using Composite Search Keys

Which indexes can you use for each of these queries?

(use = the data entries are adjacent in the index)

| | |
|---|---|
| A, AS | • age = 12 |
| AS, SA | • age = 12 and sal = 20 |
| AS | • age =12 and sal > 10 |
| None! | • age > 12 and sal > 30 |