Compotentes

Data Logical view and physical view of data. Logical view is what is seen by the programmers of the application and the physical view is what is seen by the system

People The users are the ones who pay for the system, we need to assume that they know nothing about the system (end user).

Administator is the one who charge of the database.

1.4 Procedure

We have large complex data models, but we need a way to populate it with data (need to implement).

We want to reduce the dependencies between programs and data. If we change the data it should have minimal effect on the program (UI excepted). Similarly we want very little change to data when the program is changed. This cannot be eliminated, but reduced.

Three levels (external, conceptual, internal) that each have their own schemas, called the three shema approach

external schema view for users

conceptual schema integrates external schema

internal schema defined physical storage Three schema approach: `https://en.wikipedia.org/wiki/Three_schema_approach`.

When we change the application program only the external schema should change to fit it, maybe the conceptual a bit, but not the internal schema

Language facilities

Data Definition Language (DDL) - specifies conceptual schema and subschemas and the mappings between them.

Data dictionary, data directory or system catalog - result of compilation of DDL statements in a schema; meta-data.

Data manipulation language (DML) - commands issued in a host program and preprocess or compiler interprets these commands as calls to DBMS. (SQL is default)

Query language a complete language for manipulating data interactively. Should be user friendly but not super powerful (interactive SQL is default)

Data Models

Value based model requires an attribute that can uniqly identify an object, Object based model does not.

Database Design Process

Identify functional requirements. From there figure out what data is required for those requirements, called database requirements. Structure this data in a understandable way (make a conceptual schema). This is where we make a entity relationship model. We use entity relationship models in the conceptual and logical (internal) schemas.

Entity Relationship Model

Invented in 70s and grew in popularity as variations were created, so we have a family of entity relationship models. These can get complicated and hard to read (we use a specific form of notation). An entity is something that has independent existence that is relevant to application. A single value attribute (for example a telephone) has only one value. Multivalue attributes are attributes that might have multiple parts to them (like an address). Some attributes don't belong to anyone and instead describe the relationships between them. Enities in a relationship type model dont have to have a primary key to be used.

- rectangle - entity type
- diamond - relationship type
- circles - attributes

Binary relationship is between two entity types.

Aggregation

Aggregation is the grouping of entities that have a connected relationship and only one connection outside that group to the rest of the entity map we can redraw the map with those entities grouped into one entity

Clustering

Store chuncks to data together to make accessing it faster. Try to store a bunch in memory buffer to limit the number of times we need to read.

Ordered Files

These make searching for specific files much easier (using a binary search since their order is maintained). Whenever you change the main data file all its elements need to be indexed.

B+ tree store the largest index of a block in its parent. There are three techniques, merging, splitting, and distribution. Everything is based on primary key value. The data block is represented by linked pages. Basically store the min and max values of each block in its parent block and you can just run a binary search (note: data must be sorted).

Relation Scheme

See the big ass example in lecture notes, page 85.

Relational Algebra

Relational tables require all rows to be unique as designated by the primary key (you must have at least one candidate key that is not updated or null). Foreign keys can be used to reference rows in other tables.

```
table_constraint:=
    [CONSTRAINT name]
    {UNIQUE (<column name> + ) |
    PRIMARY KEY (<column name> + ) |
    FOREIGN KEY (<column name> + )
    REFERENCES <table name> [ON {UPDATE |
    DELETE} <effect>]}
<effect> := SET NULL | NO ACTION
    (RESTRICT) | CASCADE | SET DEFAULT
```

Union operation This operation combines two tables if they have the same attributes (all the same columns) and deletes any duplicates. Denoted by $\cup$.

Set Difference This operation returns any elements in one table that are not in the other (tables must have same attributes). R-S results in values that are in R and not in S. Denoted by $-$.

Projection This operation removes or rearranges columns. R(A,B,C) $\rightarrow \pi_{C,A}(R) = R(C,A)$. Denoted by $\pi_{columns}(TABLE)$.

Selection This operations returns rows in a table that satisfy conditions. Denoted by $\sigma_{conditions}$.

Cartesian Product This operation combines two tables by taking their Cartesian product (make all possible combinations of rows). (A, B) X (C D, E F) = (A C D, A E F, B C D, B E F). Denoted by $\times$.

Rename This operation renames columns. Denoted $\rho_{table(columns)}(TABLE)$.

Natural Join This joins two tables based on a shared attribute. Think of a cartesian product but only for rows that have matching attributes. Denoted by $*$.

Binary Relational This operation is a sub operation on natural joins that provides a condition to the join to only include rows that satisfy the condition. This is equivalent to running the set operator on a cartesian product. $R * S_{\theta condition} := \sigma(R \times S)$. Denoted σ.

Intersection This operation returns any rows that are in both tables (tables must have matching attributes). Denoted $\cap$.

Views

Views are just a virtual table that compute dynamically.

To create a view:

```
CREATE VIEW <view name> (<list of column names>)
    AS <subquery> <with check option>
```

Where the subquery is the selector of what rows are in the view (views are subsets of tables). The with check option says that any operations to the view that would be invisible to the view (ie. not fit the selector for the view) wouldn't be allowed.

To delete a view:

```
DROP VIEW <view name>
```

When updating a view you run into a problem if the view's data is representative of a calculation done on the underlying table (imaging a column of sums). You cannot update this data because it would be inaccurate as updating a view doesn't touch the underlying table. There is no good way to do this aside from putting in restrictions on what you can an cannot update.

Embedded SQL

Intearctive SQL: statements are input from the terminal

Non-inteactive SQL: compiled in the application language

Statement Level Interface: application mixes in sql statements to their host language (usually have special syntax to tell sql from host language), the precompiler finds sql and translates it into calls to the host language which is then compiled by the host language compiler and executed. These can be static or dynamic sql.

Call Level Interface: sql statements are passed around as strings that are executed by the host language

Static SQL

VARCHAR: oracle data type is how we get variables. Variables that are referenced by sql statements are **host variables** and prefixed with a colon. These must be defined within an embeded sql declaration section

Cursor Operations

A cursor is created by EXEC SQL DECLARE ¡curson name¿. INSENSITIVE option creates a copy of the rows in the result set and all accesses through the cursor will be to that copy. If this is not specified then we are working with the actual row and can edit it (unless read only is declared). OPEN operation executes the query related to the cursor and we can use a couple of operations to advance it (FIRST, NEXT, PRIOR, etc). We need to specify SCROLL when declaring the cursor to use these operators, else only NEXT is allowed. We then give the cursor declaration a FOR

Update Operation

this updates the current row that the cursor is pointing at with the given values.

Some cursor update sample code

exec sql declare cursor c for ...; exec sql open cursor c; while (true) exec sql fetch c into :var ...; exec sql update ¡table¿ set ¡attribute¿ = ¡expr¿ where current of c; exec sql commit work; //commit or rollback signals the end of work

Dynamic SQL

This allows you to execute commands where you dont know the full context at the time. May have slower performance as things are stored in strings then prepared and executed instead of straight execution. we many even use dummy host variables which dont need to be declared.

If you want to execute immediately you can use EXECUTE IMMEDIATE ¡variable the statement is stored in¿. Else you prepare by processing into static sql using the PREPARE ¡statement name¿ FROM ¡statement command¿, then we execute by passing in a parameter list that will substitute for dummy values using the EXECUTE ¡statement name¿ [INTO ¡variable list¿][USING ¡parameter list¿].

Here we use cursors the same way by declaring them with OPEN.

JDBC

This is a CLI for executing SQL in a Java program. The statement is constructed at run time as a java variable which is then passed to an underlying dbms through a driver.

We connect to the data based, send statements, then process the results. See notes for sample code.

Connect to database with DriverManager.getConnection(url, id, password) if successfull it creates a connection object (called con from now on)

Create a statment object with con.createStatement(), these have executeQuery and executeUpdate abilities. Called stat.

We then assign a query string (just a string of sql) to with through stat.executeQuery(string) which returns a set of of results. Query string can have variables in it as it is constructed at run time.

We prepare the query string used above with con.prepareStatement(string). We mark placeholders with ? where actual values are plugged in using the preparedStatement's setString function (takes a index and value to put there). We can then run the executeQuery function on thsi prepared string.

We can get data from a column using getTYPE function

```
while ( res.next ( ) ) { // advance the cursor
    j = res.getString (COF_NAME); // fetch output value
    ...process output value...
}

could also have passed in the column index.
```

Schema Analysis

This is where shit gets weird.

Notation

- R, X, Y, Z - sets of attributes or relation schemes.
- r - a relation defined on a relation scheme R.
- s, t, u, v - tuples defined on some relation scheme.
- A, B, C, D, E - single attributes.
- ABC - a set consists of the attributes A, B and C.
- F, G, H - sets of functional dependencies.
- f, g, h - single functional dependencies.
- Let t be a tuple. Then t[X] denotes the X components of t.

Functional Dependencies

These are denoted as fd's. For example a relationship r **satisfies** $X \rightarrow Y$ if whenever two tuples in r agree on their X components they also agree on their Y components. Basically it denotes instances where the values in one column are dependant on another (for example all cars in the canada are tax free so the tax column relies on the country column).

$X \rightarrow Y$ is **trivial** if Y is a subset of X.

A fd g is **implied** by f (denoted by F —= g) if whenever r satisfies F it satisfies g.

The **closure** of a set of fd's F, $F^+ = \{X \rightarrow Y | F| = X \rightarrow Y\}$. This means that the closure of F (F^+) is the set of all dependencies such that F logically implies those dependencies

A fd's F is a **cover** (meaning equal to) of G (denoted $\approx$) if $F^+ = G^+$. This means that two sets of dependencies are considered equal to each other if they have the same closure (logically imply the same set of dependencies).

The **closuer** of a set of attributes X is $X^+ = \{A | X \rightarrow AisinF^+\}$. This is the set of all attributes that have a dependency to the chosen attribute (X) as defined by some dependency closure(F^+).

X is a **candidate key** of R if $X \rightarrow R \in F^+$ and no proper subset of X has this property. Translated, for a set of attributes R, an set of attributes X is a candidate key of that set if the dependency of X to R exists in the given closure of dependencies F^+ and no subset of X can fulfill this property (candidate key must be minimal).

Y is a **superkey** of R if Y contains a key of R (where Y and R are sets of attributes), basically every relation is dependent on this set of attributes.

Theorem: Given F, XY is a subset of R then:

- $F| = XY$
- Y is a subset of X^+

This means that for a set of dependencies F and two sets of attributes X and Y that are a subset of a larger set of attributes R we can say that if F logically implies that X has a dependency Y then Y is a subset of the attribute cover of X, X^+, and vice versa.

Theorem: Let $Y = A_1 ... A_n$ then $F| = X \to Y$ iff $F| = X \to A)n$ and ... $F| = X \to A_n$ This means that for a set of n attributes Y, we can only say F logically implies the dependency of X to Y if it implies the dependencies of X to each attribute in Y.

F is a **minimal conver** if

- every right side of a dependency in F is a single attribute
- there exists no $X \to A$ in F such that $F - \{X \to A\} \approx F$ (if you removed the dependency F would cover itself, ie no dependency is negligible)
- there exists no $X \to A$ in F and a proper Z of X such that $F - \{X \to A\} \cup \{Z \to A\} \approx F$ (there is no subset of X that if its dependencies were removed F would cover itself, ie no subset of X is negligible)

How to compute closer: given a set of dependencies F and a set of attributes X. First define the closure set to equal X so that it is nonempty. For each dependency in F, if the lhs is a subset of the current closure set and the rhs is not in the closure set then add the rhs to the closure set. At the end of this loop you will have computed the closure for X (X^+).

Decomposition

We can break up a large table into a series of smaller tables by creating a new table for each dependency in the main table.

An attribute A in a set of attributes R is **prime** if it is in X for some candidate key X. Basically a prime attribute is an attribute in a candidate key for a set of dependencies R (think of R as a table).

Normal Form

First Normal Form: denoted 1NF, attributes are atomic meaning they have no internal fields (at least none that the relations and schemas care about), an example of a violation of this is if you had a relation between parent and a list child (we cannot query individual children in this relation), to solve this we create an entry for each parent child relation (multiple entries for each parent)

Second Normal Form: denoted 2NF, this requires it to be 1NF and have any non-prime attributes depend on a candidate key (all of the key, not a subset), an example of a violation would be if the example solution to 1NF incorporated the gender of parents so that the parent to child relation was a candidate key (since this is unique) and the gender of parent is non-prime (cannot form a candidate key using it and anything else) but it doesn't depend on the parent-child candidate key. To fix this you need to make a new parent to gender table where this pairing works.

Third Normal Form: denoted 3NF, requires it to be 2NF and have no dependencies among any of the non-prime attributes, an example violation would be if the solution to 2NF example included a type of parent (father or mother) that is dependent on gender of parent, but both of these are non-prime keys (aren't part of candidate key). To fix this you take out the type of parent value from table and create a new table for gender to type relation.

Boyce-Codd Normal Form: denoted BCNF, created to deal with how 3NF doesn't handle overlapping candidate keys well, this requires that every instance of an attribute being fully dependent on another it is a super key.

If a table has only one candidate key then being 3NF makes it BCNF

Lossless Join

This is the concept that you should not lose any information in the decomposition of a scheme. We do this so that when we natural join the decomposition the attributes involved in the join are candidate keys for the relationships associated with the initial table. This is done to prevent generating new rows in the joined table that weren't in the original. For example R=(A, B, C, D, E) with dependencies A→BC, CD→E, B→D, and E→A, and we break it into R1 = (A, B, C) and R2 = (A, D, E). This is a lossless join because we join R1 and R2 on A, and A→BC which we can expand trivially into A→ABC which is the same as A→R1. Basically the decomposition must be joinable onto a set of attributes that can imply one of the decomposed relations based on the initial relations functional dependencies. You can test this using the **chase algorithm**.

Chase Algorithm

First you create a table (called tableau). There is a row for each dependancy and a column for each relation. For each cell put the name of the attribute if it is involved in the corresponding relation, else give the cell a unique marker (usually the attribute name with a counter). Then decompose all dependencies so that each has only one attribute on the right. Iterating through each dependency, try to match column, where rows have matching values on the left of a dependency they should have matching values for the right side. Try to get a row that has all unscripted values, if you can do this the algorithm terminates and you have a lossless join. The prof executes this using ϕ for all scripted values and a for unscripted ones and you try to convert an entire row to a's.

see `http://stackoverflow.com/questions/23596464/lossless-join-and-decomposition-from-functional-dependencies` for details

Dependency Preserving

When decomposing things we need to watch for violating any dependencies. Basically list all dependencies that each relation can hold and the cover of the original relation must be a subset of the union of them.

Example Problems

R(A, B, C, D, E), F = A→B, BC→E, ED→A

1List all keys

ED→A $\implies$ so by identity ED→ADE $\implies$ we know that A→B so ED→ABDE $\implies$ then add C to get CED→ABCDE $\implies$ CDE is a key.

BC→E $\implies$ so by identity BC→BCE $\implies$ we can add BCD→BCDE $\implies$ we know that ED→A so BCD→ABCDE $\implies$ BCD is a key.

A→B $\implies$ so by identity A→AB $\implies$ we can add AC→ABC $\implies$ we know that BC→E so AC→ABCE $\implies$ we can add D ACD→ABCE $\implies$ ACD is a key.

1NF: yes, because no attributes are collections

2NF: yes, because all attributes are prime (all attributes are in one of the candidate keys)

3NF: yes, because all attributes are prime

BCNF: no, because A→B but A is not a super key(since there are instances of attributes not being dependent on A)

Transaction Management

Just like anything else the database can crash, for instance if you attempt to read data that doesn't exist yet. Concurrency really fucks with this. The **dirty read problem** is when one transaction reads data from a row that has been modified but not committed (reading stale data). **Inconsistent analysis problem** is when a transaction reads the same data multiple times and it keeps changing due to another transaction committing changes to it.

`https://technet.microsoft.com/en-us/library/aa213029(v=sql.80).aspx`

ACID

- Atomicity - transaction performed in its entirety or not at all
- Consistency - keep the database consistent (dont crash it)

- Isolation - concurrently acting transactions shouldnt interfere with each other
- Durability - committed data should not be lost

The **schedule** is the order that transactions will be executed in (to monitor concurrent transactions).

A **cascading rollback** is the database rolling back all transactions that relied on a transaction that caused a failure (this can snowball quite far, hence the name cascading). A schedule avoids cascading rollback if it only reads items that were fully committed by transactions (then you only have to ever rollback on transaction).

Concurrency Control

A **serial schedule** has each transaction run to completion before beginning the next. If a nonserial schedule has the same effect as a serial schedule it is **serializable**.

Serializability

A serializable schedule has the same effect as if you finished each transaction completely before moving on. Two schedules are **view equivalent** if:

- the set of transactions are the same
- if one reads the initial value so does the other (before it gets fucked with)
- the value written by one transaction will read the same for each transaction
- a transaction that writes the final value for each transaction will write the same value

A vies is **serializable** if is view equivalent to a serial schedule.

`http://www.tutorialspoint.com/dbms/dbms_transaction.htm`

Read before Write Model A transaction must read data before it can be written and cannot have a read without a write. Assume that some function has been executed between the read and write.

Conflicts Two steps in a schedule are conflicting if they are in different transactions on the same data and one is a write. These might produce different effects on the database if you swapped their order of execution.

Test for serializability:

1. create a precedence graph
   (a) add node for each transaction
   (b) add edge where two transactions contain conflicting steps from the transaction that has the first step in the conflict
2. if there is a cycle it is not serializable
3. else find an ordering via topological sort
   (a) start with node with no incoming edges, remove it and all outgoing edges
   (b) repeat until empty
4. output is now a serial schedule

Locking

You have RLOCK that prevents others from writing that data and WLOCK which prevents others from reading it. Call UNLOCK when done. Livelock and deadlock are problems.

- Prevention
  - get all locks at start
  - order lock requests
- Detection
  - draw a wait for graph
    - node for each transaction
    - edge from A to B if A is waiting to lock an item held by B
    - if a cycle exists deadlock has happened
    - kill one transaction to fix

The test for serializability is exactly the same as earlier just with locks instead of actions.

Two Phase Unlocking is a protocol stating that all locks must precede all unlocks (lock down everything first), this ensures serializability

Isolation Levels

In relational databases we access all items that satisfy a predicate which makes it much harder to control concurrency so we need different isolation levels. We run into the problem of **phantom updates**. This is when the order of transactions can cause two sequential reads to be different. Locking down a row in a table does not prevent an update from adding rows that would fit the predicate. To fix this you can use predicate locking. These, of course, can conflict.

Under two phase lock a transaction doesn't release until it commits or aborts. We can weaken the lock to increase concurrency. This brings about various isolation levels:

- read uncommitted(lowest) - dirty reads are allowed
- read committed - only write locks must be held until end of transaction so that all data is committed right before a read of it, prevents dirty reads, but consecutive reads may differ
- repeatable read - this holds locks until end of transaction but predicate locks are not managed so phantom updates can occur
- serializable(highest) - this is the isolation level described above where locks are required to be released at end of transaction

Lock Types Locks can be on rows/predicates, read/write, short/medium/long. Short locks execute a statement then release and long are held until end of transaction. Write locks are handled the same at all isolation levels, long locks are for update/insert/delete. Read locks differ with level:

- read uncommitted - no read locks (a transaction can read a write locked item allowing dirty reads)
- read committed - short duration read on rows and predicates (prevents dirty reads)
- repeatable read - long duration read on rows and short duration on predicate
- serializable - long duration read on rows and predicates

Cursor Stability This is a spin of of read committed where the read lock on a row access through a cursor is medium duration, it is held until the cursor moves

Crash Recovery

Storage types:

- volatile: data is lost easily
- nonvolatile: slightly more reliable
- stable: data is assumed to be permanent

Failure types:

- Transaction-local: only one transaction fucked up and only one needs to be rolled back
- system-wide: some or all transactions need to be rolled back
- media: cannot recover, data lost

Logical file: file as seen by programmer (set of logical records)

Block atomic unit of storage, also called a page (may contain one or more logical records)

When you open a file a buffer the size of the file is allocated. Blocks are moved to the buffer by input or output commands. Fuck with it by read or write commands. The input/output command just take a parameter telling it what record to access, the read/write commands take a parameter describing the record to access and one for the variable containing the information you want to use.

0.0.1 Recovery Schemes

Log based Technique keep a log file of sequential changes to the system and keep it on stable storage (for every change store a record containing the changed values in it). This way we can defer updates in the log until we can be sure it won't fuck shit up. You can also do stuff with immediate updates by letting everything through and keep a log of the state at a time and if a failure occurs and rollback if required.

Media Failure Recovery Whenever no transactions are happening dump the database into stable storage incase failure.

0.1 Security

Security protects data against unauthorized access. Create accounts, assign/remove privilege, assign security level.

Account level privilege allos the creation of stuff, the relation level allows us to control people's privileges (via AUTHORIZE key word). Each relation in database has an owner (person who created it) who can pass privileges to other users (using GRANT or REVOKE then the list of functions they are allowed to execute ON then teh list of columns they can touch TO username).