Database Management Systems

Author: Guanzhou (Jose) Hu 胡冠洲 @ UW-Madison CS564

Teacher: Paris Koutris

Figures included in this note are from Prof. Koutris's slides unless otherwise stated.

Introduction

This course focuses on two parts:

1. User's view on using a database management system (DBMS)
2. Designing, implementing, and maintaining a DBMS

A **Database** (DB) is an *organized* collection of *structured* data. A **Database Management System** (DBMS) is software that:

- Automate operations on data in a DB
- Boost performance of data storage/retrieval
- Safely allow concurrency
- Scale up / out to very large databases
- Protect from system crashes

Some terms used in the database world:

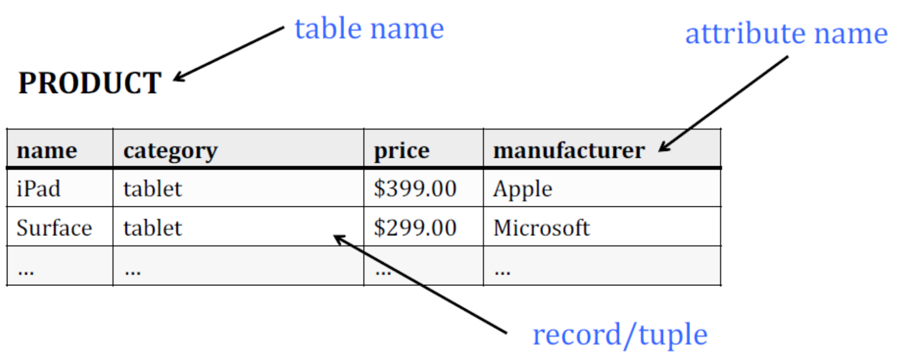
- **Data model** = abstract layout of data, e.g., table, array, key-value, graph, ..
- **Schema** = a concrete *type* of a data model, e.g., a SQL table structure saying "the 1st column is ID, the 2nd column is Name, ..."
- **Instance** = a data record, a concrete *value* of a schema
- **Query** language = high-level language allowing user interactions with a DBMS
    - Types:
        - *Declarative*: e.g., SQL, Relational Calculus
        - *Procedural*: Relational Algebra
    - *Data Independence*: a desired feature of a query language that application does not need to change when underlying data models change

Data Models

Data model is the most important layer of *abstraction* we will learn in this course.

Relational Model

A **Relational** data model organizes data as a collection of *tables* (*relations*). A table looks like:



name	category	price	manufacturer
iPad	tablet	$399.00	Apple
Surface	tablet	$299.00	Microsoft
...	...	...	...

- Every *column* is an *attribute*
- Every *row* is a data *record (*tuple*)
- A *domain* is the valid range of an attribute's value
- Every attribute has an *atomic* type, i.e., non-composite type

The **SQL** (*Structured Query Language*) is a query language over a relational-model database to query & manipulate relational data.

- Details about the SQL language can be found in the slides, in [this tutorial](), and in course activities notebooks; Some notes from slides #2, #3, #4:
    - Null values, `NOT NULL` constraints, null propagation, `IS NULL`
    - 3-valued logic
    - Multiset semantic & using `DISTINCT`
    - Set semantic & using `UNION ALL`
    - Compute maximum - `ORDER BY` with `LIMIT 1` vs. nested check
    - Foreign key constraint: reject vs. cascade update/delete vs. set null
    - Multiple `FROM` sources: inner join by `,`, outer joins
    - Correlated subqueries execute multiple times

    - Set comparison - `ANY`, `ALL`
    - `COUNT(DISTINCT attr)`
    - Aggregation with `GROUP BY` & `HAVING`
- A good point of using a query language is that it hides implementation details of the DBMS. Users only need to follow a standard set of *semantics* - and the DBMS takes care of translating the query commands into actual high-performance query operations

Entity-Relationship (E/R) Model

Entity-Relationship (E/R) model is a data model describing what information a database should hold and the relations between them. It is a good visual starting point when designing schemas for a given scenario. We can then easily convert an ER model to a relational model for actual SQL coding.

See the ER model section below.

Key-Value Model

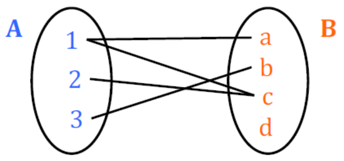
A *key-value* model (sometimes called *KV*, *NoSQL*) is a new data model gaining popularity, given the fact that traditional relational databases have poor performance on some modern workloads. This course will not go into details about KV stores.

Entity-Relationship (E/R) Model

An ER model is a collection of entity sets and their relationships:

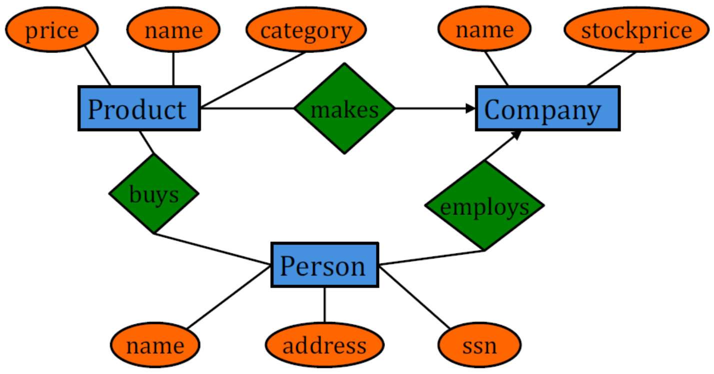
- Every object instance is an **entity**, i.e., an object distinguishable from other objects
- An **entity set** (*rectangle*) represents a collection of entities of the same category
- Each entity set has a set of **attributes** (*oval*) attached to it
- A **relation** R (diamond) is a subset of the cartesian product of two entity sets $A \times B$, e.g.,

– **A** = {1, 2, 3}, **B** = {a, b, c, d}
– **R** = {(1, a), (1, c), (2, c), (3, b)}



> Note: Here, 1, 2, 3, and a, b, c, d are entities (instances) of that entity set.

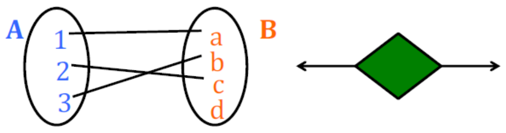
An example of an ER model:



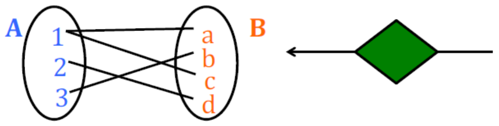
Relationships

A relation R can have one of the following types of *multiplicity*:

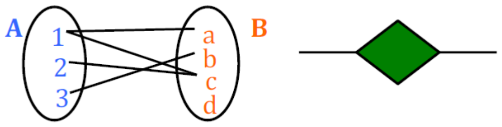
1. *One-one*:



E.g., a UW student one-one maps to a CS login account.

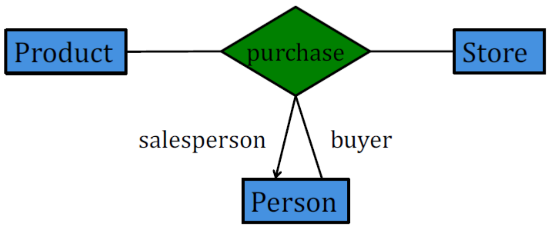
2. *Many-one*: 

E.g., a company 1 can manufacture two products a & c, but a product can only be manufactured by one particular company.

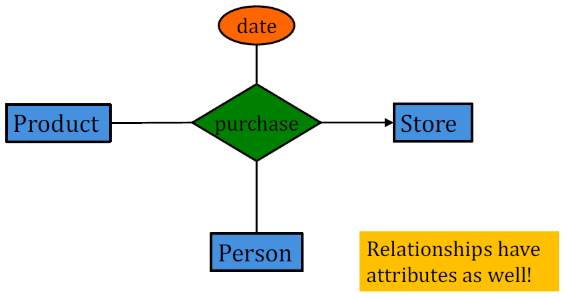
3. *Many-many*: 

E.g., a person 1 can by two products a & c, and a product c can be bought by two persons 1 & 2.

A relation can also be *multi-way*, where it is the cross product of more than two entity sets. You can also label the edges to indicate *roles*: collapsing two entity sets into one when we need the same entity set multiple times.

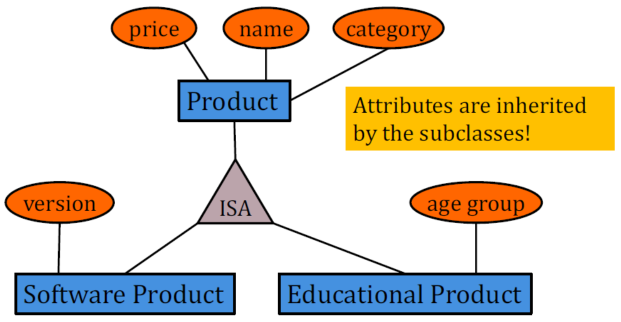


A relation can have attributes as well:



Subclasses

An entity set can have *subclasses*, i.e., entity sets that inherit all its attributes (and probably having their own extra attributes). Subclasses of an entity set are expressed through a *triangle*.

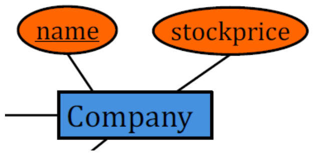


Constraints

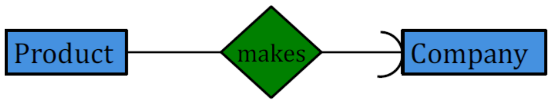
A *constraint* is an assertion about an ER model that must be true at all times. It's good to have as much constraints as possible when drawing an ER diagram.

There are several types of constraints:

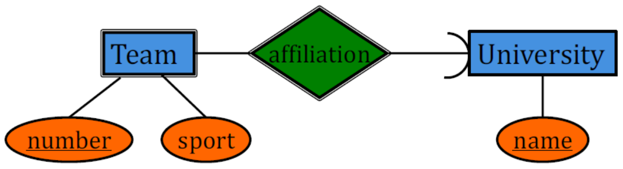
- *Key* (<u>underline</u>): an attribute uniquely specifies an entity object



- *Single-value*: an entity has at most one value for a given attribute or relationship
    - That attribute of this entity set has a single value
    - A many-one relation implies a single value constraint
- *Referential integrity* (*round arrow*): a relationship is many-one, is single-value, and it must exist for every entity in the entity set on the "many" side



An entity set is called *weak* when its complete key attributes come from other classes to which they reference, for example the team entity set here:



To design a beautiful E/R model, remember:

- Avoid redundancy
- Keep it simple
- Make something an entity set ONLY IF:
    - It is more than a name; it has at least one non-key attribute, OR
    - It is the "many" in a many-one / many-many relation
- Avoid weak entities; give unique key identifier to every entity set

Converting ER to Relational Model

ER model is good for representing application requirements, while relational model is better for efficient computation. So we first design an ER schema then convert that to a relational schema. Translation rules:

- Entity set E
    - Normal: $\rightarrow$ New table with key attributes as primary key + its own non-key attributes as normal columns
    - Weak: $\rightarrow$ New table with its own key attributes AND all referenced sets' key attributes as primary key + its own non-key attributes as normal columns
- Relationship R
    - Many-many: $\rightarrow$ New table with ALL key attributes from connected entity sets as primary key + the relation's own attributes as normal columns
    - Many-one / one-one: $\rightarrow$ Give the "many" side table a new column of the primary key of the "one" side as a *foreign key* + the relation's own attributes as new columns
- Subclasses:
    - Option 1: Each entity set as a separate table, each having all of that class's attributes
    - Option 2: Each entity set as a separate table; Parent has its all attributes; Children have only primary key + subclass-specific attributes
    - Option 3: Only having the parent entity set table with all attributes, including all subclass attributes, and leave a column null when you don't need it

Functional Dependencies (FD)

By now, we have figured out how to use E/R model to model an application and then transform the E/R diagram to a relational schema to build a relational database. In this section, we show how to use **functional dependencies** to refine (normalize, decompose) a "bad" relational schema.

If two tuples agree on attributes $A = \{A_1, \dots, A_n\}$ then they MUST agree on the attributes $B = \{B_1, \dots, B_m\}$, then we say A *functionally determines* B (i.e., $A \rightarrow B$).

Example: $\text{SSN} \to \text{name, age}$ in

SSN	name	age	phoneNumber
934729837	Paris	24	608-374-8422
934729837	Paris	24	603-534-8399
123123645	John	30	608-321-1163
384475687	Arun	20	206-473-8221

FD Properties

Functional dependency has the following properties:

- If $A, B \to C, D$, then C & D are *independently* determined by A, B
  - We can *split* this FD into two FDs: $A, B \to C$ and $A, B \to D$
  - We CANNOT do the same thing to the left hand side
- An FD $\mathbf{X} \to A$ is *trivial* if A belongs to the attributes set $\mathbf{X}$
- *Armstrong's Axioms*:
  - *Transitivity*: If $A \to B$ and $B \to C$, then $A \to C$
  - *Reflexivity*: For any subset $\mathbf{S} \subseteq \{A_1, \dots A_n\}$, we have $\{A_1, \dots A_n\} \to \mathbf{S}$
  - *Augmentation*: If $X \to Y$, then $X, Z \to Y, Z$

An FD is *domain knowledge*, meaning an inherent property of the application, NOT a temporary thing we can infer from a set of tuples. Given a table of a set of tuples:

1. We can confirm that a certain FD is definitely invalid
2. We can say an FD seems to be valid
3. We can NEVER prove that an FD is valid: there might be violating tuples not included in this table

Keys are special cases of FDs: $K \to$ all other columns in a relational schema.

Closure Algorithm

Its natural to think about closures under this notation:

- For a set of FDs F, its *closure* F^+ is the set of all FDs logically implied by F
- For an attributes set $\mathbf{X}$, its *closure* $\mathbf{X}^+$ is the set of all attributes $\mathbf{B}$ such that $\mathbf{X} \to \mathbf{B}$

Example:

- Given **Product**(name, category, color, department, price)
  - $name \longrightarrow color$
  - $category \longrightarrow department$
  - $color, category \longrightarrow price$
- We have $\{\text{name}, \text{category}\}^+ = \{\text{name}, \text{color}, \text{category}, \text{department}, \text{price}\}$

Computing the closure of an attribute set is straightforward, much like the ϵ-closure algorithm in programming languages:

```
Let $X = \{A_1, A_2, \dots, A_n\}$
UNTIL $X$ doesn't change REPEAT:
    IF $B_1, B_2, \dots, B_m \longrightarrow C$ is an FD AND
        $B_1, B_2, \dots, B_m$ are all in $X$
    THEN add $C$ to $X$
Output $X$
```

To compute the closure F^+ of a set of FDs F:

1. For each FD $f : \mathbf{A} \to \cdots$
   1. For each subset A of $\mathbf{A}$, compute A^+ based on F
   2. Then for each subset $B \subseteq A^+$, $A \to B$

Superkeys & Minimal Basis

A *superkey* is a set of attributes $\mathbf{A} = A_1, \dots, A_n$ such that this set functionally determines any other attribute B in the relational schema.

- I.e., $\mathbf{A}^+ =$ the full set of attributes

- A *key* is a *minimal superkey* where none of its subsets can determine all other attributes

Similarly, a *minimal basis* of a set of FDs F is S iff:

- $S^+ = F^+$
- Every FD $\in S$ has exactly one attribute on the right hand side
- If we remove any FD from S, then S^+ is no longer F^+
- If for any FD $\in S$ we remove one or more attributes from the left hand side, then S^+ is no longer F^+

To compute the minimal basis of F, follow the algorithm:

1. Split the right hand side into only one attribute
2. Remove all redundant FDs which can be logically implied by remaining ones
3. Clean up remaining FDs' left hand side

Schema Normalization

We now use FDs to help *decompose* and *normalize* "bad" schemas.

Decomposition

We can *decompose* a relation $R(A_1, \ldots, A_n)$ by creating two relations $R_1(B_1, \ldots, B_m)$ and $R_2(C_1, \ldots, C_k)$, where $\{B_1, \ldots, B_m\} \cup \{C_1, \ldots, C_k\} = \{A_1, \ldots, A_n\}$. We then say the instance of R_1 is the projection of R onto the dimensions $B_1, \ldots, B_m$.

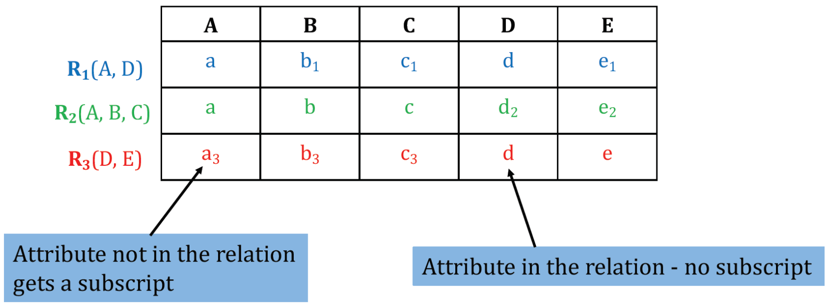
A good schema decomposition should achieve:

- Minimize redundancy
- Avoid information loss (*Lossless-join*): For any instance R, joining (recovering) the decomposed relations will give $R' = R$
- Preserve all the FDs (*Dependency preserving*)
- Ensure good query performance

Lossless-Join & Chase Algorithm

The *Chase* algorithm for checking lossless-join decomposition:

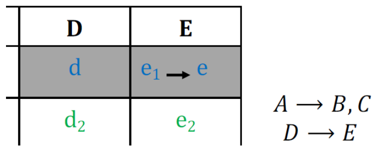
1. Create a table with attributes of the original relation and one row for each relation we split into



	A	B	C	D	E
R_1(A, D)	a	b_1	c_1	d	e_1
R_2(A, B, C)	a	b	c	d_2	e_2
R_3(D, E)	a_3	b_3	c_3	d	e

Attribute not in the relation gets a subscript
Attribute in the relation - no subscript

2. At each iteration, for every FD, check:
   - If in a row, the left hand side has no subscription and the right hand side does, then remove the subscription
   - If the left hand side has different subscription with the right hand side, then make the right hand side's subscriptions to be the left hand side's



D	E
d	$e_1 \rightarrow e$
d_2	e_2

3. Repeat until no more updates.
   - If there is a row without any subscription, then we can say the decomposition is lossless-join

A	B	C	D	E
a	b	c	d	e
a	b	c	d_2	e_2
a_3	b_3	c_3	d	e

- Otherwise, it is not

Dependency Preserving

Given R and a set of FDs F, the procedure to determine dependency preserving split:

1. Compute the closure F^+ of the FD set F
2. For each split relation, say R_1, then its FD set $F_1 =$ all non-trivial FDs in F^+ with attributes in R_1
3. If by forcing all FDs F_i in all split relations R_i, we can recover all FDs F in R, then we say this split is dependency preserving

A bad example which is NOT dependency preserving:

R(A, B, C)
- $A \longrightarrow B$
- $B, C \longrightarrow A$

R_1

A	B
a_1	b
a_2	b

R_2

A	C
a_1	c
a_2	c

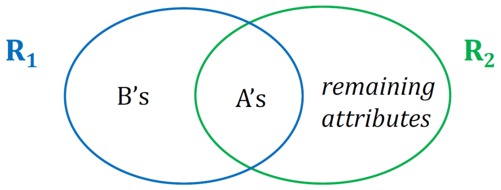
Boyce-Codd Normal Form (BCNF)

A relation R is in *Boyce-Codd normal form* (BCNF) if: whenever $X \to B$ is a non-trivial FD, then X is a superkey of R. Equivalently, it means that for every attribute set X, either $X^+ = X$ (trivial), or $X^+ =$ all attributes (superkey).

- Example: three attributes, only FD is $\text{SSN} \to \text{name}, \text{age}$, and key $= \{\text{SSN}\}$
- Every binary relation is in BCNF

To decompose a relation into all BCNF relations, follow:

1. Find an FD $A_1, \ldots, A_n \to B_1, \ldots, B_m$ that violates BCNF condition
2. Split R into R_1 and R_2 as:



3. Repeat until no BCNF violations can be found

Such BCNF decomposition has the following properties:

- Removes certain redundancy
- Is always lossless-join
- NOT always dependency preserving; example is the same as above section

Third Normal Form (3NF)

A relation R is in *third normal form* (3NF) if: whenever $X \to A$, one of the following is true:

- $A \in X$ (it is trivial), or
- X is a superkey, or
- A is part of some key of R (we say A contains *prime attributes*)

Notice that BCNF implies 3NF. Every BCNF relation is in 3NF.

It is always possible to lossless-join & dependency-preserving decompose a relation into 3NF! The algorithm:

1. Apply BCNF decomposition until all relations are in 3NF
2. Compute the minimal basis F' of F

3. For each non-preserved FD $X \to A$ in F', add a new relation $R(X, A)$
    - If all FDs are already preserved, then R is indeed in BCNF

Relational Algebra

Relational Algebra is a mathematically formal way of representing queries to a relational database. It is different from *relational calculus* because relational algebra is more like an operational, procedural representation of how to compute.

- Operands: relations or variables representing relations
- Operators: common computation we need to do with relations
- Inputs & Outputs are relational instances, but the schema is fixed for a given expression
- In SQL we use *multisets* (multiple rows could be exactly the same), but in relational algebra we consider relations as *sets* (meaning NO duplicate rows)

Basic Operators

Basic operators in relational algebra:

- *Selection*: $\sigma_C(R)$
    - C is a condition; can be a composite one; CANNOT have subqueries in C
    - Outputs the rows of R that satisfy C; output schema is the same as input schema
    - `SELECT * FROM table WHERE condition;`
- *Projection*: $\pi_{A_1,...,A_n}(R)$
    - Outputs only the columns $A_1, \ldots, A_n$ from R; output schema is $R'(A_1, \ldots, A_n)$
    - MUST remove any duplicate tuples, because the output instance is a set as well
    - `SELECT DISTINCT A1, A2 FROM table;`
- *Union*: $R_1 \cup R_2$
    - Inputs schemas should be the same
    - Outputs the unioned rows from both; output schema is the same
    - Will remove duplications in the output
- *Difference*: $R_1 - R_2$
    - Inputs schemas should be the same
    - Outputs the rows in R_1 but NOT in R_2; output schema is the same
- *Cross Product*: $R_1 \times R_2$
    - Cartesian product of all tuples in R_1 with all tuples in R_2
    - Say input schemas are $R_1(A_1, \ldots, A_n)$ and $R_2(B_1, \ldots, B_m)$ and they do not overlap, then output schema is $R'(A_1, \ldots, A_n, B_1, \ldots, B_m)$
    - `SELECT * FROM t1, t2;`
- *Renaming*: $\rho_{A_1,...,A_n}(R)$
    - Say input schema is $R(B_1, \ldots, B_n)$, it will be renamed to output $R'(A_1, \ldots, A_n)$
    - This operator is important for name-based relational algebra (as opposed to positional)
    - `SELECT b1 AS a1, b2 AS a2 FROM table;`

Operators are compositional, but logically equivalent expressions MAY represent different procedures of computation:

- $\pi_{SSN,Age}(\sigma_{Age>24}(\text{Person}))$: select, then project
- $\sigma_{Age>24}(\pi_{SSN,Age}(\text{Person}))$: project, then select

Derived Operators

Derived operators in relational algebra, which can be derived from basic operators:

- *Intersection*: $R_1 \cap R_2 = R_1 - (R_1 - R_2)$
    - Input schemas should be the same
    - Outputs the intersected rows from both; output schema is the same
    - `SELECT t1.a, t1.b FROM t1, t2 WHERE t1.a = t2.a AND t1.b = t2.b;`
- *Theta Join*: $R_1 \bowtie_\theta R_2 = \sigma_\theta(R_1 \times R_2)$
    - Essentially, a cross product with condition θ
    - `SELECT * FROM t1, t2 WHERE theta;`
- *Equi-Join*: $R_1 \bowtie_\theta R_2$

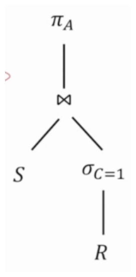
    - Theta join where the condition θ only contains equalities between attributes
    - `SELECT * FROM t1, t2 WHERE t1.a = t2.b;`
- *Natural Join*: $R_1 \bowtie R_2 = \pi_{SSN,Age,Dept}(\sigma_{SSN=dSSN}(\text{Person} \times \rho_{SSN \to dSSN}(\text{Department})))$
    - Equi-Join on all the *common* attributes of the two tables
    - Input schemas may have common attributes, and output schema will be the set of all attributes, keeping only one copy of each common attribute
        - If inputs do not have any common attribute, natural join is cross product
        - If input schemas are the same, natural join is intersection
    - `SELECT SSN, Age, Dept FROM Person, Department WHERE Person.SSN = Department.SSN;`
- *Semi-Join*: $R_1 \ltimes R_2 = \pi_{A,B,C}(R_1 \bowtie R_2)$
    - Natural Join followed by a projection on the attributes of R_1
    - Essentially, removes the rows in R_1 that does not join
- *Division*: $R_1 / R_2 = \pi_A(R_1) - \pi_A(\pi_A(R_1) \times R_2 - R_1)$
    - Suppose $R_1(A,B)$ and $R_2(B)$, the output will contain all values a such that, for EVERY tuple (b) in R_2, tuple (a,b) is in R_1
    - Say input schemas are $R_1(A_1, \dots, A_n, B_1, \dots, B_m)$ and $R_2(B_1, \dots, B_m)$, then output schema is $R'(A_1, \dots, A_n)$

Extended RA Syntax

There are still some syntaxes in an extended RA:

- Simple *Aggregate*: $\gamma_{Agg(Y)}(R)$
    - Agg is an aggregation operation (sum, count, min, max) on attributes Y
    - Output schema is $R'(\text{a numeric val})$
    - `SELECT COUNT(y) FROM table;`
- *Group-By Aggregate*: $\gamma_{X,Agg(Y)}(R)$
    - Group by attributes X
    - Agg is an aggregation operaation on attributes Y
    - Output schema is $R'(X, \text{a numeric val})$
    - `SELECT x, COUNT(y) FROM table GROUP BY x;`
- Some more not used in this course

Any RA expression can be expanded in the form of an expression tree. For example, $\pi_A(\sigma_{C=1}(R) \bowtie S)$ expands to:

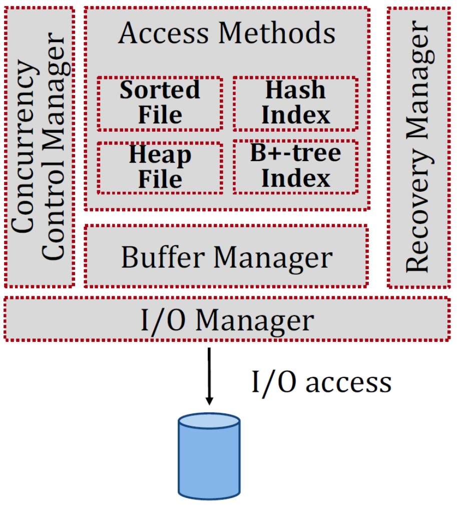


- Relational algebra has limited *expressive power*. For example, *transitive closures* (*recursions*) cannot be expressed by RA
- Anything related to "ordering" in SQL language also cannot be expressed by RA, e.g., `LIMIT`, `ORDER BY`, `ROW NUMBER`

Data Storage

Starting from this section, we are entering the second phase of this course: DBMS implementation.

General architecture of a storage manager:



I will skip the parts involving:

- Disk hardware anatomy, Flash SSD anatomy, and sequentiality
- Basic ideas of memory hierarchy and caching

Because they have been explained in detail in OS courses.

Buffer Manager

The cache used in a DBMS for caching data in main memory is often called a *buffer manager*.

- Actual disk blocks are called *pages*
- Slots in the buffer are called *frames*; supports *reading*, *flushing*, and *releasing*

Each frame bookkeeps:

- *Pin count*: number of current users of the page in this frame; only pin count = 0 frames are considered available (some what different from classical caching, where in-use cache lines can still be evicted)
    - *Pining* on request, pin count `+= 1`
    - *Unpinning* on release, pin count `-= 1`
- *Dirty bit*: for write-back cache, indicates modified page; write I/O is *lazy* - it happens when some page replaces the dirty page

Cache eviction policies are those common ones used, e.g., LRU, FIFO, CLOCK, MRU, ...

- LRU: when pin count reaches 0, push to the tail of the LRU queue (essentially, a FIFO in classic caching)
- CLOCK: when pin count reaches 0, flip the reference bit; when fetching an available frame, the first time we arrive at an available frame with reference bit = 1, we clear it to 0 and move on (essentially, a *second-chance* FIFO)

The sequential scan problem is termed *sequential flooding* here.

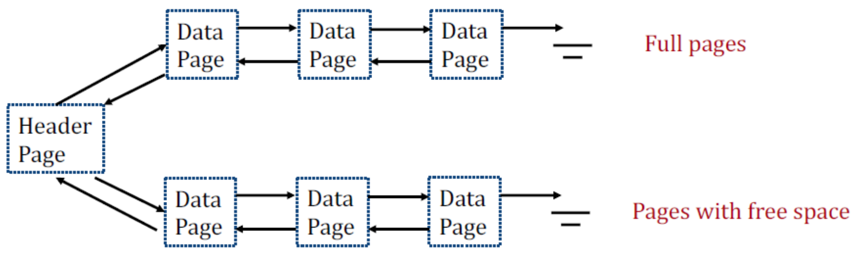
File Organization

To store data in files, we need to consider:

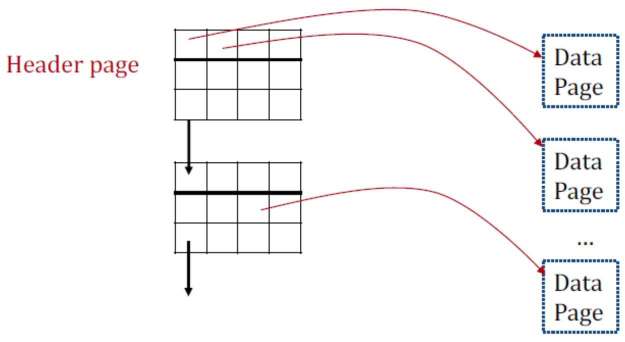
- How to organize pages within a file?
- How to organize records within a page?
- How to organize data within a record?

We can store pages as an *unordered* (*heap*) file. Each page has its unique ID (*pid*) and each record has its unique ID (*rid*). A heap file can be organized as different layouts:

- Two *doubly-linked lists*:



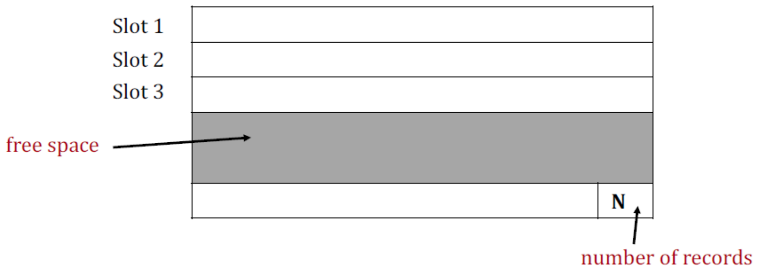
- *Page directory*:



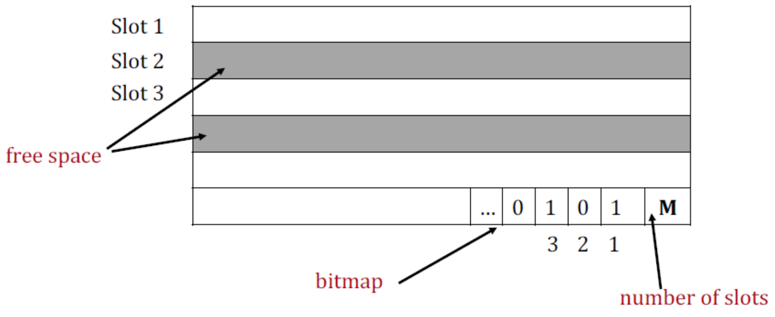
Page Organization

A page is a collection of record slots. An *rid* = <*pid*, slot number within the page>. A page can be formatted as:

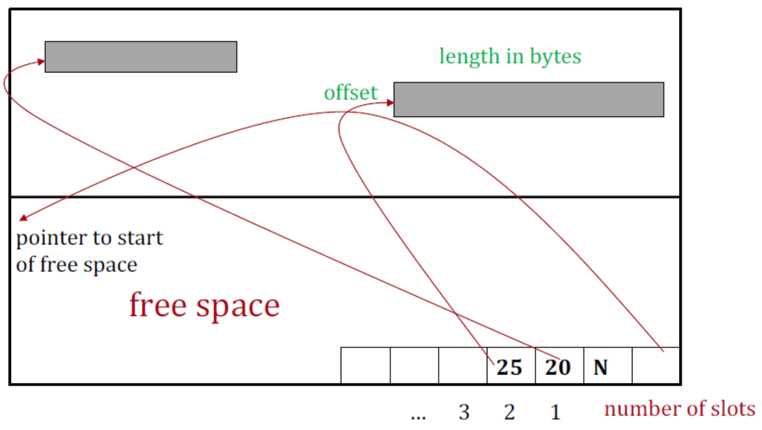
- Fixed-length records: each slot has a fixed length
    - *Packed* organization: first N slots for the N records; remaining slots are free space; last slot for storing the current number of records N



        - Insert to the next free slot
        - Delete will require moving the last record to fill the empty gap - that record's *rid* will change
    - *Unpacked* organization: last slot for storing a *bitmap* of slots + the total number of slots M in this page



        - Insert looks up an empty slot in the bitmap
        - Delete will require clearing a bit in the bitmap
- Variable-length records:

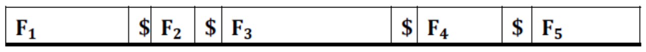


- Maintains a slot list at the end of the file
- Deletion sets the offset to -1
- Insertion to the start of free space; if no enough space is available, we need to reorganize (to remove fragmentation)
- The *rid* of a record keeps unchanged when we move the record at reorganization, since it is defined by the slot number in the slot list

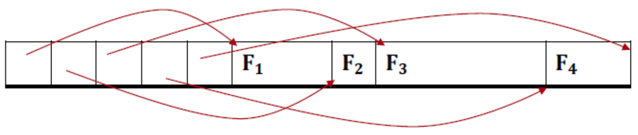
Record Format

The number of fields + type of each field (i.e., column) will be stored in a common system catalog. To organize fields in a record, we have the following choices:

- Fixed-length fields
- Variable-length fields:
  - Use *delimiters* to denote separation of fields - need a scan of the whole record to locate a field



  - Use an array of *offsets* at the beginning of the record



The problem with this *row-store* format is that it is very inefficient when we want to `SELECT` a column from a table - a common workload in SQL! To address this problem, to can instead store the table *vertically*:

- Each column of a relation is stored in a different file
- Each split "record" in a file will then only have one field
- This is called the *column-store* design

Index Structures

Alternative to *unordered* heap files, there are many other organizations which fit other access patterns than sequential scans. We can have *sorted* files to speedup range queries, but the insertion and deletion overhead will be very bad.

Modern DBMS use various types of *indexes*:

- Hash table
- B+ tree
- Bitmap index
- ...

Indexing Basics

An **index** is a data structure that organizes records of a table to speed up lookups. The attribute or combination of attributes we use to lookup the records is called the *search key*. A search key can be *composite*, which means a combination of attributes.

In the index structure, every *entry* can be either:

- The full record data itself, i.e., the index structure itself is a file organization of records - at most one index on a given table can use this choice
- <Key k, *rid* / pointer of the record data> - better for large data records

Every file can have multiple indexes on different search keys. An index can be classified as:

- *Primary* (contains the primary key) vs. *Secondary* (otherwise)
    - In a primary index, there are no duplicates for one value of search key
    - If the search key contains a unique key, then it is called a *unique* index - also no duplicates
- *Clustered* (order of entries matches the physical storage order of records, often implied by the full-record-data index storage choice) vs. *Unclustered*

An index structure has the following properties:

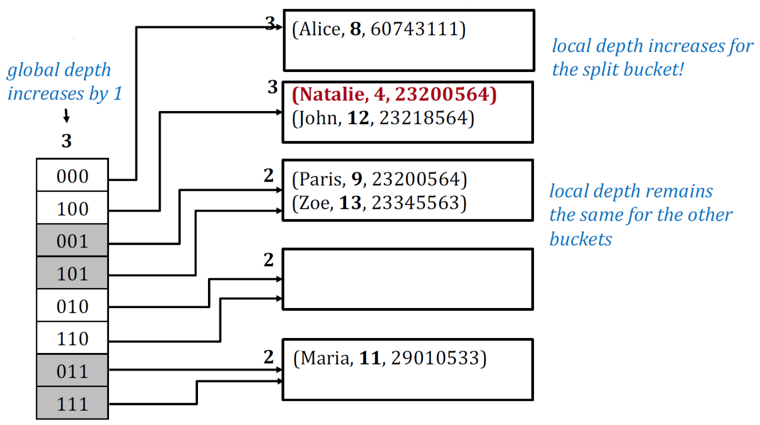
- Supported access types:
    - Equality search
    - Range query
- Time to lookup (access, search) a record
- Time to insert a record
- Time to delete a record
- Space complexity of the structure

When deciding creating indexes on tables, we should consider attributes in `WHERE` clauses to be the candidates for search keys and the choice of index structure should match what search conditions we are going to use (e.g., single-point equality for hash tables, range conditions for B+ trees).

Hash Tables

Hash tables are efficient for single-point equality queries but not suitable for range queries. Hash tables have (in expectation) constant time for both search and insert.

- Static (*closed-address*): bucket as linked list of primary page + *overflow* pages
    - Ideal hash function is *uniform* - low *load factor*
    - *Skew* distribution of keys over buckets is bad
    - Table may need to be resized - expensive operation
- Extendible (dynamic resizing)
    - Use the last *global depth* bits of the binary form of search key to locate bucket
    - Every "bucket" is a pointer to a page, which has a fixed capacity
    - Every page has a *local depth*, originally the same as global depth
    - Whenever a bucket gets full on insertion:
        - If it has a local depth equal to global depth:
            1. Increase global depth by 1
            2. Split the overflowed bucket by new global depth bits
            3. Local depth increases for the split bucket; Remains the same for other buckets - they don't need to split at this time
        - If it has a local depth smaller than global depth, it means some buckets have been split at this level - just split the overflowed bucket and increment the local depth



- On deletion:
  - (optional) Can choose to coalesce with its sister page if they fit in one
  - If the bucket becomes empty:
    1. Remove the empty page
    2. Update the directory to point to its sister page
- If the full directory fits in memory, then only one I/O for each lookup

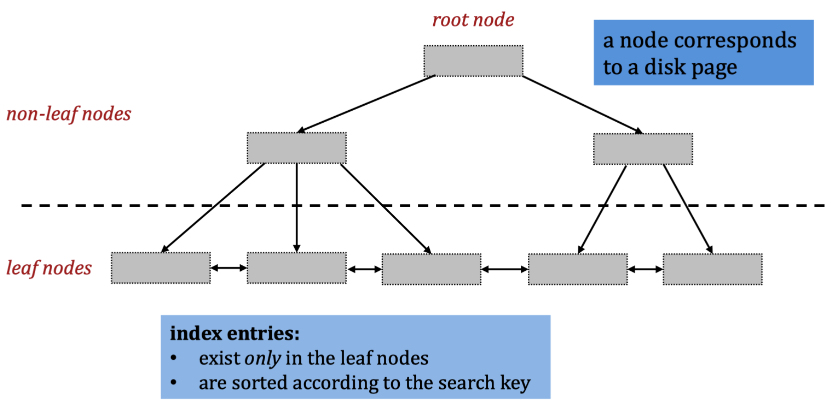
- *Open-address*: not mentioned in this course

The ratio of actual storage size over meaningful record data is called the *fudge factor* f, typically ~1.2.

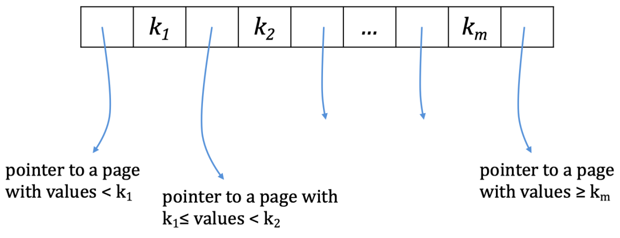
B+ Trees

B+ trees are great indexing structures for range conditions. A *B+ tree* is a *self-balancing* tree structure similar to AVL and red-black trees, but not binary. It is widely used in file systems and DBMSs.

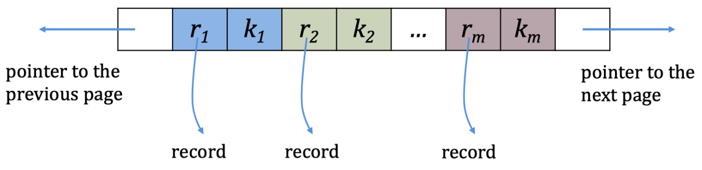
Basic structure of a B+ tree:


root node
a node corresponds to a disk page
non-leaf nodes
leaf nodes
index entries:
• exist only in the leaf nodes
• are sorted according to the search key

- B+ tree has an *order* of d; each node of a B+ tree contains m entries, where $d \leq m \leq 2d$
  - The root node can have $1 \leq m \leq 2d$ entries
  - A non-leaf (internal) node with m entries has $m+1$ pointers to lower-level nodes in different ranges:


k_1
k_2
...
k_m
pointer to a page with values < k_1
pointer to a page with $k_1 \leq$ values < k_2
pointer to a page with values ≥ k_m

  - A leaf node with m entries has m pointers to data records (*rids*), along with two pointers to previous & next page



- Every leaf node resides in its own page on storage

- Search (point query & range query):
  1. Start from root node, examine internal nodes to traverse until the correct leaf node
  2. Check that leaf node:
     - If doing equality search, done;
     - If doing range search, traverse the leaves sequentially using previous/next pointers

  Cost model for point search:

  $$\text{I/O cost} = h - L_B + 1 + I$$

  - h is height,
  - L_B is the number of levels fit in buffer
  - $I = 0$ if clustered, otherwise 1

  Cost model for range query:

  $$\text{I/O cost} = h - L_B + P_{leaf} + P_{record}$$

  - P_{leaf} is the number leaf pages we read
  - P_{record} is the number of record result pages we read
- Insert:
  1. Find the correct leaf node L like in searching
  2. Insert data entry into L:
     - If L has enough space, done;
     - Else, split L into two leaf nodes L and a new node L' to the right, redistributing entries evenly:
       1. Copy up the now leftmost key of L' into the correct place of the parent node P
       2. Add the pointer to L' into P as well
       3. If P now exceeds the capacity limit $2d$, recursively propagate the procedure upwards; If the root node exceeds its capacity limit, we will eventually split the root node into two and add a new root node atop, increasing the tree height by 1
- Delete:
  1. Find the correct leaf node L like in searching
  2. Remove the entry from L:
     - If L is at least half-full, done;
     - Else, L falls down to only $d - 1$ entries, we try to borrow an entry from neighbor sibling nodes
       - If any of the two neighbors have $> d$ entries, borrow one and done;
       - Else, we have to *merge* L and siblings:
         1. If merge occurs, must also delete an entry from parent P
         2. This can lead to P being less than half-full, so the procedure may recursively propagate upwards; If merging the second-topmost level and the root node only has 1 entry, merge with the root node, decreasing the tree height by 1

There are several common metrics people use to measure B+ tree costs:

- *Fan-out* f: number of pointers to child nodes coming out of a non-leaf node
  - $f = (2d + 1) \cdot F \in (d + 1, 2d + 1)$
  - Typically assumed as constant as the upper bound is limited by $2d + 1$
- *Fill-factor* F: the percentage of available slots in the B+ tree that are filled
  - $F < 1$
  - Typically around $\frac{2}{3}$
- *Height* h: number of levels of non-leaf nodes
  - $h = \lceil log_f N \rceil \geq 1$, where N is the number of leaf pages
  - The number of leaf pages = number of records / (page capacity $\cdot F$)

- Higher fan-out means smaller height & less cost per search
- Typically around 3 or 4

Internal levels often do not take much space and can be kept in the cache for quick access.

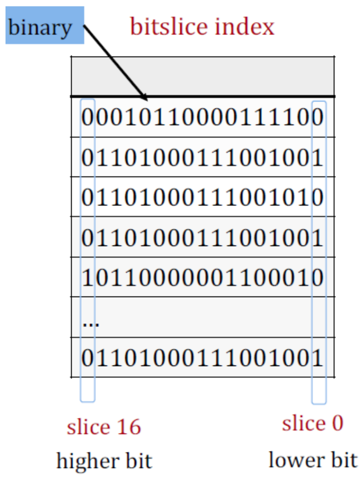
Bitmaps & Bitslices

Bitmaps are an efficient way for storing values with a small domain.

- One bitmap per possible value in domain + NULL value
- To efficiently map *rid*s to bit positions, we layout bitmap pages so that *pid*s are sequential
- More space-efficient than a B+ tree when size of value domain $<$ data entry size of B+ tree

$$\text{Index size} = \#\text{records} \cdot (\text{domain\_size} + 1)\ bits$$

Bitslices are a way to apply bitmaps on values with larger domain, such as integers.



- Convert integer values into fix-length binary representation, then for each bit position (*slice*), use a bitmap to store it
- Querying a bitslice for values $\leq$ threshold:
  1. Convert the threshold value also to binary representation
  2. Create a result bitmap for storing query results
  3. Start from the highest bitslice, compare data record bit with threshold bit:
     - If threshold bit is 0 and data record bit is 1, put 0 in result bitmap
     - If threshold bit is 1 and data record bit is 0, put 1 in result bitmap
     - Otherwise, cannot decide yet for this data record, leave blank and need to consult the next lower bitslice
  4. If there are any blanks in the result bitmap, move to the next lower bitslice
- Aggregation with bitslices, e.g., `SUM`:
  1. Count the number of 1's in slice 15 and multiply the count with 2^{15}
  2. Count the number of 1's in slice 14 and multiply the count with 2^{14}, add to the result
  3. Repeat...

Supporting SQL Operators

In this section, we are gonna see how to use the buffer manager and storage indexing structures to efficiently support actual SQL workloads.

External Sorting

External sorting is the variant of sorting where not all data can fit in the main memory, so we must interact with external storage devices. External sorting happens when users issue `ORDER BY` on a big table, in B-tree bulk loading, or in some join algorithms. It is essential to all DBMS systems.

We first start with *external merging* where:

- We want to merge two sorted lists, one with M pages and the other with N pages
- We want to merge them into one big sorted list of $M + N$ pages, with the fewest number of I/Os - $2(M + N)$
- We have a buffer of 3 frames = # input lists + 1

The algorithm:

1. Read the two smallest pages from each list, merge into the remaining empty frame until it becomes full;
2. Write the merged frame to disk, then continue merging until we finished any of the two input pages;
3. Read the next page of that list (whose first page has been exhausted), and repeat merging.

$$\text{I/O cost} = 2(M + N)$$

Using the external merging algorithm, we can build the *external merge-sort* algorithm:

1. Split the table into chunks small enough to sort in memory (called *runs*, size B each)
2. Merge the runs using external merging
3. Every sorting of run / merging of runs is called a *pass*: pass 0 is the sorting to get initial runs, and subsequent passes merge the runs

Say we have an input table of N pages and a buffer of B frames, the best I/O cost of external merge sort is

$$\text{I/O cost} = 2N(\lceil \log_{B-1} \frac{N}{B} \rceil + 1)$$

where in the first pass we make $\frac{N}{B}$ runs and then, in every subsequent pass, we merge $B - 1$ of the runs. Every pass takes $2N$ I/Os to finish.

> An optimization to external merge sorting is *replacement sorting*, which creates initial runs of average size $2B$, instead of fixed size of B. See slides #16, page 33.

Using the index structures, the buffer manager, and a method to do large sorting, we now look into how to implement relational operators with the support of the I/O layer.

- Logical operators: union, selection, projection, join, grouping, ... (SQL semantics)
- Physical operators: nested loop join, sort-merge join, hash join, index join, ... (How are they implemented by a DBMS)

Selection Operator

The essential notion for selection operator implementation is *access path*:

- *File scan*: scan entire file to search for the record; I/O cost = number of pages N
- *Index scan*: if an index is already available on some predicate that is usable; I/O cost depends on the index
  - Hash index on equality predicates only: $O(1)$ if no duplicates
  - B+ tree index: $h - L_B + 1 + X$, where h is height, L_B is number of index pages in buffer, and X depends on whether the index is *clustered*:
    - Unclustered: $X =$ number of selected tuples in the worst case
    - Clustered: $X =$ number of selected tuples $/$ number of tuples per page

We say an index **matches** a selection predicate if the index can be used to evaluate the selection.

- A hash index on composite key $(A, B, \ldots)$ matches a selection if all attributes in the composite key appear in the selection predicate with equality
- A B+ tree index on composite key $(A, B, \ldots)$ matches a selection if a set of attributes in the predicate form a prefix of the composite key, with any operations
- A predicate can match more than one index; For conjunction (ANDs), we can choose to:
  - Use one index, then check remaining conditions for retrieved tuples
  - Use multiple indexes, and intersect them
- For disjunction (ORs):
  - Use one index, then check remaining conditions for retrieved tuples
  - If some indexes cover all the conditions, where we union them and then retrieve the records

Selectivity of an access path means the percentage of records in relation that satisfy the selection condition and retrieved by the access path. We generally want to choose the most selective (minimal selectivity) path. See slides #17, page 23~26 for estimating selectivity.

Projection Operator

Simple case is without `DISTINCT` where we just scan the file and proejct out the attributes. Hard case is with `DISTINCT` where we need to remove duplicates:

- *Sort-based*: scan the file and project out attributes, sort the resulting set of tuples, then scan it again to discard adjacent duplicates
- *Hash-based*: first project out attributes and split the results into $(B-1)$ partitions using a hash function, then read each partition into memory and use an in-memory hash table to remove duplicates
- *Index-based*: if the projection attributes are a prefix of the search key of some ordered index structure, we can retrieve the index entries in order, and simply discard unwanted fields and remove duplicates along the way

Cost estimation, see slides #17, page 30~38.

Join Operator

Algorithms for equi-joins:

- *Nested loop join*: naive approach

  ```
  for page P_R in R:
      for page R_S in S:
          join tuples in P_R with tuples in P_S
  ```

  - I/O cost = M_R + $M_S \cdot M_R$, where M_R is the number of pages in R
  - Outer relation should better be the smaller one
  - Uses minimum number of buffers - 3 frames suffice
- *Block nested loop join* (BNLJ):

  ```
  for block of (B-2) pages from R:
      for page P_S in S:
          join tuples in block with tuples in P_S
  ```

  - I/O cost = $M_R + M_S \cdot \lceil \frac{M_R}{B-2} \rceil$
- *Index nested loop join* (INLJ): suppose S has an index I on the join attribute

  ```
  for page P_R in R:
      probe each tuple of P_R in I to retrieve matching tuples
  ```

  - I/O cost = $M_R + |R| \cdot I^*$, where I^* is the I/O cost of searching the index
- *Block sorted index nested loop join*: combining block grouping with an ordered index I for S

  ```
  for block of (B-2) pages from R:
      sort the tuples in the block
      probe each tuple of the block in I to retrieve matching tuples
  ```

  - I/O cost = $M_R + (sort(B-2) + I'_{B-2}) \cdot \lceil \frac{M_R}{B-2} \rceil$, where $sort(B-2)$ is the cost to sort $(B-2)$ frames of tuples and I'_{B-2} is the cost to sequentially match $(B-2)$ frames of tuples from the ordered index
- *Sort merge join* (SMJ): sort R and S on the join attribute using external sorting, then read the sorted relations into buffer in order and merge
  - If a relation has already been sorted on the join attribute we can skip the first step for it
  - Can also handle multiple matches in the merge phase, but in this case merging may take worse than linear cost ($O(M_R \cdot M_S)$)
  - I/O cost = $sort(R) + sort(S) + M_R + M_S$ in no duplicate case and $sort(R) + sort(S) + M_R \cdot M_S$ otherwise
  - An optimization is to generate sorted runs of size $2B$ for R and S, and do multi-way merge of all sorted runs; memory requirement is $B^2 \geq max(M_R, M_S)$ and the I/O cost can be as low as $3(M_R + M_S)$
- *Hash join* (HJ): partition R and S into $(B-1)$ partitions using a hash function h, then join each partition of R with the corresponding partition with the same hash value of S using BNLJ
  - I/O cost = $3(M_R + M_S)$ in best case if all buckets from one of the two relations have size $\leq B-2$
  - Suppose a perfectly uniform hashing result, then if $B^2 \geq min(M_R, M_S)$ we can get an I/O cost of as low as $3(M_R + M_S)$ due to the second phase BNLJ can finish in two passes
  - If $min(M_R, M_S) \leq B-2$, then hash join needs only one pass

For more general join conditions:

- Multiple equalities: BNLJ always applicable; INLJ applicable if having index on the composite key or individually cover all keys; SMJ and HJ now sort or hash using the combination of attributes

- Inequality conditions: BNLJ always applicable; INLJ needs clustered B+ tree; SMJ and HJ not applicable

> For set operations, intersection is a special case of a join, and union and difference are similar to how we remove duplicates in projection. For aggregation operations, just sorting cost.

Aggregation Operation

For aggregation operations like `SUM`, `COUNT`, `MIN`, `MAX` used with `GROUP BY`:

- *Sort-based*: sort, then scan the result produce min or max on the first seen entry for a new group
- *Hash-based*: hash on group-by attributes, record running aggregate at hash table entry, and output the hash table
- *Index-based*: if we have an ordered index available on aggregate attributes
  - Without grouping: simply use the index
  - With grouping: we must have the group-by attributes in the search key as well; if they form a prefix of the search key, then the entries can be retrieved in group-by order

Query Optimizer

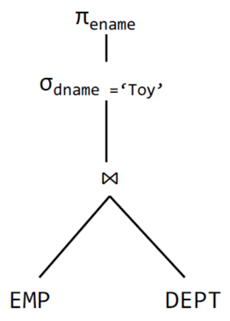
A query optimizer sits in between the SQL language interface and the storage manager. It first parses SQL query to relational algebra formula, then optimizes the RA formula, generates actual *query plans* (annotated RA trees), and estimates the cost.

Annotated RA Trees

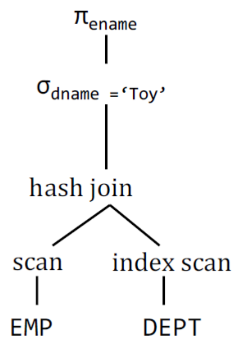
A SQL query can be translated into an RA expression, hence an RA tree, which represents the order of operations to be performed to complete this query. RA tree can be annotated with algorithm names to represent the actual plan of query execution.

Example: for the SQL query `SELECT DISTINCT ename FROM Emp E, Dept D WHERE E.did = D.did AND D.dname = 'Toy'`:

- A possible RA tree:



- A possible partially annotated RA tree for this tree:



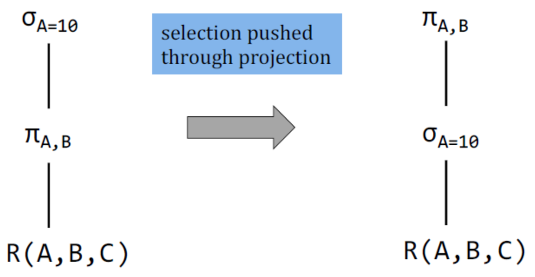
Note that for the same SQL query, there can be many equivalent RA trees representing different orders of operations, and hence even more possibilities for annotated RA trees.

Optimization Plans

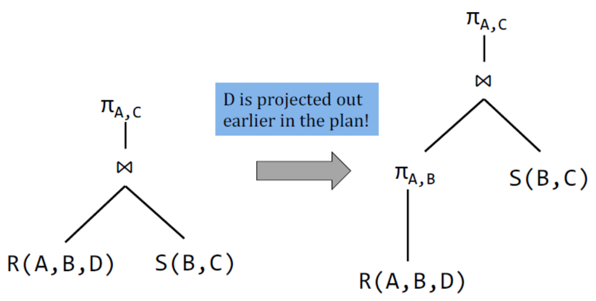
The space of all possible annotated RA trees is huge, so it's impractical to explore all of them and pick the best one. There are several heuristic rules for the optimizer to make better decisions:

- Pushing down selections

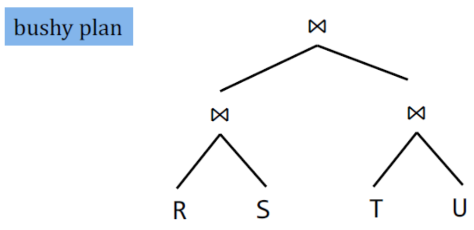
    - Always possible to push a selection through another selection
    - Might be possible to push a selection through a projection; not possible when the selection involves attributes from multiple relations



- Pushing down projections



- Reordering joins
    - Rules:
        - Commutativity: $R \bowtie S \equiv S \bowtie R$
        - Associativity: $(R \bowtie S) \bowtie T \equiv R \bowtie (S \bowtie T)$
    - *Left-deep joins* are easy to but not efficient
    - *Bushy joins* better than cascading joins as we can do lower level joins in parallel:



Besides reordering operations, there is also opportunity to apply *pipelining*:

- We can always choose to *materialize* intermediate results to disk before starting the next operation in tree
- If the next operation is a scan-like operation (e.g., row-by-row projection), we can pipeline the computation: Whenever we generate a tuple at this level we can feed it to the next operator

Cost Estimation

It is helpful to estimate the cost of a query plan:

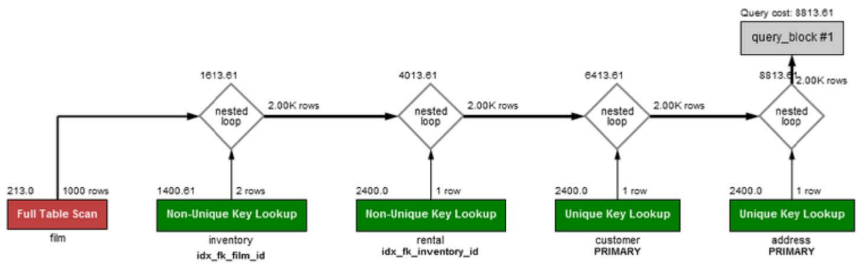
- Estimating the size of intermediate results
- Estimating the cost of each operation step in the plan; depends on:
    - Input cardinalities
    - Complexity of the algorithm

See slides #19 page 24~ for cost estimation examples.

Estimation cannot be 100% accurate. Production DBMS systems store statistics in the *system catalog* and use them to guide the estimation, e.g.,:

- Number of tuples (cardinality) of input relations
- Size in pages
- Number of distinct keys
- Range of values

This is a hard problem and is still an active research area in DBMS field. A real-world example looks like:



Transaction Management

DBMS should also provide the support for transactions to be fault tolerant and consistent across failures.

Definition of Transactions

A **transaction** is a sequence of DBMS operations (e.g, SQL statements) bounded together into a single *atomic* unit. Example: a bank account transfer composes of "{subtract $10 from A, add $10 to B}". The whole transaction either succeeds or not at all - we cannot leave the DBMS state as partially committing a transaction.

```
BEGIN TXN;
    UPDATE account
        SET balance = balance - 10
        WHERE account_no = 1;
    UPDATE account
        SET balance = balance + 10
        WHERE account_no = 2;
COMMIT;
```

Transactions are easy to guarantee on a single-node system, but much harder to do correctly for a distributed DBMS. Please see my distributed systems note for more on this. Distributed transactions =

- Concurrency control: *pessimistic* vs. *optimistic*
- $+$ Atomic commits: 2PC, ...

The "ACID" Principle

A canonical DBMS should provide the following four properties for transactions, summarized as the "**ACID**" principle:

- **A**tomicity: all or none, cannot commit partial result of a transaction
  - *Commit*: all the changes are made
  - *Abort*: no changes are made, as if nothing happened
- **C**onsistency: called *external consistency* in the DBMS scenario (not the data consistency in distributed replicated systems), means to obey application-specific invariants
  - E.g., bank account balance must be ≥ 0
  - A DB in a consistent state will remain in a consistent state after a transaction
    - The programmer makes sure the transaction logically takes a consistent state to a consistent state
    - The DBMS makes sure the transaction is actually atomic
- **I**solation $\equiv$ *Serializability*: transactions cannot see each other's intermediate results, but only complete (commited) transaction results
  - If T_1 and T_2 are interleaved, the results should be the same as either first T_1 then T_2, or first T_2 then T_1
- **D**urable: once committed, the results should be persistently stored and survive across failures

There are several challenges to make a DBMS "ACID":

- Power failures (*fail-stops*)
- Users may manually abort the program - *rolling back* the log
- Concurrent users - locking
- Ensuring ACID while maintaining good performance

Write-Ahead Logging (WAL)

The *log* is an ordered list of modifications of the form: `<TXNID, location, old-data, new-data>`. It should record both *redo* & *undo* information for every update to the DBMS state.

The log itself will be duplexed on persistent storage BEFORE the flushing of actual data records (*write-ahead* logging, WAL; think of a journaling FS).

- The log record for a data page update is persisted before the data page update (for atomicity)
- A transaction is commited only after all of its log records have been persisted (for durability)

> Improvement to WAL: see the ARIES protocol as an example.

Transaction Concurrency

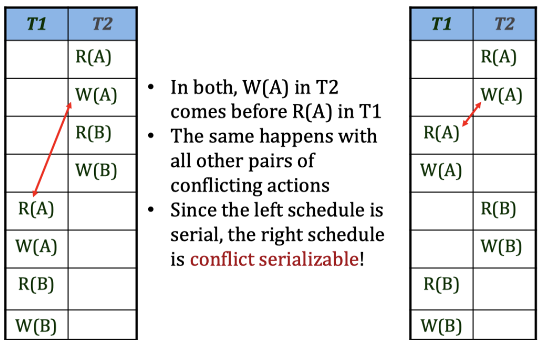
The DBMS typically executes multiple transactions concurrently for better multiplexing and hence better performance. It can create a *schedule* for a set of concurrent transactions:

- *Serial* schedule: T_1, then T_2; or T_2, then T_1; Different serial schedules may generate different results, depending on the order, but they are all "correct" in the sense of ACID
- *Serializable* schedule: operations interleaved, but the outcome is the same with some serial schedule

Two operations are said to *conflict* if they come from different transactionis, involve the same variable, and at least one of them is write. Conflicts can lead to *anomalies*:

- Dirty read: T_1 reads intermediate data modified by T_2 before T_2 commits (caused by W-R conflict)
- Unrepeatable read: T_1 reads data twice, but the data is modified by T_2 in the middle (caused by R-W conflict)
- Overwriting: T_1 writes to a place modified by T_2 but T_2 has not commited yet (caused by W-W conflict)

Two schedules are *conflict equivalent* if they involve the same actions of the same transactions, and every conflict is ordered in the same direction if the two schedules. A schedule is *conflict serializable* if it is *conflict equivalent* to some serial schedule. A conflict serializable schedule indicates that it is serializable.



T1	*T2*
	R(A)
	W(A)
	R(B)
	W(B)
R(A)	
W(A)	
R(B)	
W(B)	

- In both, W(A) in T2 comes before R(A) in T1
- The same happens with all other pairs of conflicting actions
- Since the left schedule is serial, the right schedule is conflict serializable!

T1	*T2*
	R(A)
	W(A)
R(A)	
W(A)	
	R(B)
	W(B)
R(B)	
W(B)	

The *conflict graph* of a schedule maps each transaction to a node and adds an edge between $T_1 \rightarrow T_2$ if there is a conflict in the direction of T_1 to T_2

- A schedule is conflict serializable IF AND ONLY IF its conflict graph is a DAG (directed acyclic graph), i.e., it has at least one *topological ordering*
- Then, this schedule is serializable as if executed in serial order of that topological order

For resolving mutual exclusion, locking is used. There will be *S* locks (shared locks, reader locks) and *X* locks (exclusive locks, writer locks). The *two-phase locking* (2PL) protocol is used to enforce and guarantee a conflict serializability schedule. Strict 2PL goes:

- Lock on a variable is acquired when we first meet that variable in the execution of this transaction
- All locks are released all at once only after the transaction has commited

> Optimized 2PL says that locks can be gradually released in the second phase (but never re-acquired).

We could have *deadlocks* with 2PL. We need to detect/prevent deadlocks with some mechanisms (covered in OS courses).