Databases

Muchang Bahng

Fall 2024

Contents

1 Theory of Relations **5**
1.1 Structure . . . 5
1.2 Operations and Relational Algebra . . . 5
1.2.1 Set Operations . . . 6
1.2.2 Projection and Selection . . . 7
1.2.3 Product and Join . . . 7
1.2.4 Renaming . . . 10
1.2.5 Monotonicity . . . 11
1.3 Constraints . . . 11
1.3.1 Domain Constraints . . . 12
1.3.2 Key Constraints . . . 12
1.3.3 Referential Integrity . . . 13
1.4 Functional Dependencies . . . 13
1.4.1 Structure on Spaces of Functional Dependencies . . . 14
1.4.2 Projections of Functional Dependencies . . . 15

2 Design Theory for Relational Databases **17**
2.1 Anomalies and Decomposition . . . 17
2.1.1 Boyce-Codd Normal Form . . . 18
2.1.2 Recovery and Chase Test . . . 21
2.2 The Entity-Relationship Model . . . 24
2.3 Relationships and Multiplicity . . . 25
2.3.1 Multiplicity of Binary Relationships . . . 25
2.3.2 Multiplicity of Multiway Relationships . . . 28
2.4 Subclasses of Entity Sets . . . 29
2.5 Weak Entity Sets . . . 30
2.6 Translating ER Diagrams to Relational Designs . . . 32

3 SQL **37**
3.1 Structure and Constraints . . . 37
3.1.1 Nulls . . . 38
3.1.2 Modifying Tables . . . 39
3.1.3 Constraints . . . 40
3.2 Basic Relational Algebra Operations . . . 42
3.2.1 Renaming . . . 43
3.2.2 Set/Bag Operations . . . 43
3.2.3 Predicates, Projection, Selection . . . 44
3.2.4 Products and Joins . . . 45
3.3 Arithmetic and Sorting . . . 47
3.4 Aggregate Functions . . . 48

3.4.1 Group By . . . 49
3.4.2 Having . . . 50
3.5 Nested Queries and Views . . . 52
3.5.1 Subqueries . . . 52
3.5.2 Scalar Subqueries . . . 53
3.5.3 Quantified Subqueries . . . 54
3.5.4 Views . . . 54
3.6 Recursion . . . 55

4 Index and B+ Trees 56
4.1 Hardware and Memory Layout . . . 56
4.2 Search Keys and Index . . . 58
4.3 Tree Index . . . 60
4.3.1 B-Trees . . . 60
4.3.2 B+ Trees . . . 61
4.3.3 Clustered vs Unclustered Index . . . 66
4.4 Hash and Composite Index . . . 66
4.5 Index Only Plans . . . 68
4.6 Exercises . . . 69

5 Query Processing and Optimization 72
5.1 Brute-Force Algorithms . . . 72
5.1.1 Nested Loop Joins . . . 72
5.2 Sort-Based Algorithms . . . 75
5.2.1 External Merge Sort . . . 75
5.2.2 Sort Merge Joins . . . 77
5.2.3 Zig-Zag Join . . . 79
5.2.4 Other Sort Based Algorithms . . . 80
5.3 Hash-Based Algorithms . . . 80
5.3.1 Hash Join . . . 80
5.3.2 Other Hash Based Algorithms . . . 82
5.4 Exercises . . . 82
5.5 Logical Plans . . . 85
5.5.1 Query Rewrite . . . 85
5.5.2 Search Strategies . . . 86
5.6 Physical Plan . . . 88
5.6.1 SQL Rewrite . . . 89
5.6.2 Cardinality Estimation . . . 89
5.7 Exercises . . . 90

6 XML 93
6.1 XPath . . . 95
6.2 XQuery . . . 98
6.3 Converstion of XML to Relational Data . . . 101

7 JSON 104

8 Transactions 109
8.1 Consistency: Serizability and Precedence Graph . . . 111
8.2 Isolation and Concurrency Control . . . 113
8.2.1 Locks . . . 113
8.2.2 Snapshots . . . 115
8.2.3 Timestamp-Based Concurrency Control . . . 116
8.2.4 Conflicts and Isolation Levels . . . 116
8.3 Atomicity and Durability . . . 117

8.4 Logging 117
8.4.1 Checkpointing 119

9 Big Data **123**
9.1 Parallel Databases 123
9.2 Map-Reduce 123

This is a course on database languages (SQL, XML, JSON) and database management systems (Postgres, MongoDB).

Definition 0.1 (Data Model)

A **data model** is a notation for describing data or information, consisting of 3 parts.

1. *Structure of the data.* The physical structure (e.g. arrays are contiguous bytes of memory or hashmaps use hashing). This is higher level than simple data structures.
2. *Operations on the data.* Usually anything that can be programmed, such as **querying** (operations that retrieve information), **modifying** (changing the database), or **adding/deleting**.
3. *Constraints on the data.* Describing what the limitations on the data can be.

There are two general types: relational databases, which are like tables, and semi-structured data models, which follow more of a tree or graph structure (e.g. JSON, XML). We'll cover in the following order:

1. The theory of relational algebra.
2. Practical applications with SQL.
3. Theory and practice of XML.
4. Theory and practice of JSON.

1 Theory of Relations

The most intuitive way to store data is with a *table*, which is called a relational data model, which is the norm since the 1990s. We will first talk about the structure, then the operations, and finally some constraints.

1.1 Structure

Definition 1.1 (Relation)

A **relation** is a set or multiset R with the following data structure.

1. Each element $r \in R$ is called a **tuple**/**row**, with the form $r = (a_1 \in T_1, \ldots, a_n \in T_n)$. Despite its name and our use of indices, the tuple itself is *not ordered*.
2. Each T_i is a primitive[a] type (e.g. int float, string, but not a list or set).[b]
3. Each T_i is given an alphanumeric name A_i as a variable, called the **attribute**. We denote $\mathbf{A}$ as the set of the attribute of relation R.
4. The **schema** of R tells us a summary of its structure: $R(A_1\ T_1, A_2\ T_2, \ldots, A_n\ T_n)$.
5. Given a time parameter t, R_t represents the **instance** of the relation R at time t.

Given these names, we can visualize relations as tables, tuples as rows, and attributes as columns.

[a] loosely defined here

[b] Note that this is also a form of constraint on our data, but the types are usually defined under structure.

We introduce this in this abstract way because there are many implementation differences between DBMSs. For example, the set of available primitive types may differ, or we may be able to define attribute names with special characters. Even in *temporal databases*, we may also keep track of the history of its instances rather than just the current instance.

Example 1.1 (Schemas)

Here are a few schemas, which has the name of the relation followed by the attributes and its types.

```
Beer (name string, brewer string)
Serves (bar string, price float)
```

Note that this is analogous to a function declaration in C++.

Definition 1.2 (Compatiblity)

Two relations R and S are **compatible**, denoted $R \simeq S$, if they have the same schema (same set of attribute types and names).

While we have not talked about any types yet, there is a common one that we should mention now.

Definition 1.3 (Null)

A **null** value, denoted with ω indicates an unknown but not empty value. The specific behavior of this value will be covered when we get into SQL implementations.

1.2 Operations and Relational Algebra

We have just defined sets, and the natural thing to do is to construct functions on sets, i.e. what operations are legal. We introduce this with *relational algebra*, which gives a powerful way to construct new relations from given relations. Simply put, an algebra is an algebraic structure with a set of operands (elements) and operators.[1]

Definition 1.4 (Relational Algebra)

A **relational algebra** consists of a set of operations O grouped into 4 broad categories.

1. *Set Operations.* Union, intersection, and difference.
2. *Removing.* Selection removes tuples and projection removes attributes.
3. *Combining.* Cartesian products, join operations.
4. *Renaming.* Doesn't affect the tuples, but changes the name of the attributes or the relation itself.

Let's take a look at each of these operations more carefully, using the following relation.

bar	beer	price
The Edge	Budweiser	2.50
The Edge	Corona	3.00
Satisfaction	Budweiser	2.25

Figure 1: The example relation, which we will denote `serves`, which we will use to demonstrate the following operations.

1.2.1 Set Operations

Definition 1.5 (Union)

Given $R \simeq S$, $R \cup S$ is the set/multiset of tuples in either R or S.

Definition 1.6 (Intersection)

Given $R \simeq S$, $R \cap S$ is the set/multiset of tuples in both R or S.

Definition 1.7 (Difference)

Given $R \simeq S$, $R - S$ is the set of tuples in R but not S, or the multiset that keeps track of counts in R and S.

Note that this set is not minimal.

Lemma 1.1

Given $R \simeq S$,

$$\begin{aligned} R \cap S &= R - (R - S) && (1) \\ &= S - (S - R) && (2) \\ &= R \bowtie S && (3) \end{aligned}$$

Proof. The natural join will check for all attributes in each schema, but sine we assumed that they had the same schema, it must check for equality over all attributes.

[1] Not the technical definition.

Example 1.2 (Symmetric Difference of Relations)

Let $R(A,B,C) = \{(1,2,3),(4,2,3),(4,5,6),(2,5,3),(1,2,6)\}$ and
$S(A,B,C) = \{(2,5,3),(2,5,4),(4,5,6),(1,2,3)\}$
Compute $(R-S) \cup (S-R)$ (symmetric difference). Identify one tuple in the result.

1. $(4,5,6)$ This tuple appears in both R and S, so it's not in the symmetric difference.
2. $(2,5,3)$ This tuple appears in both R and S, so it's not in the symmetric difference.
3. $(\mathbf{1},\mathbf{2},\mathbf{6})$ This tuple appears only in R, so it's in R - S and thus in the symmetric difference.
4. $(1,2,3)$ This tuple appears in both R and S, so it's not in the symmetric difference.

1.2.2 Projection and Selection

Definition 1.8 (Selection)

The **selection** operator σ_p filters the tuples of a relation R by some condition p.

$$\sigma_p(R) \tag{4}$$

It must be the case that p is deducible by looking only at that row, but it may not be (e.g. the condition where the count of an attribute in the row passes a threshold).

Definition 1.9 (Projection)

Given that $L \subset \mathbf{A}$ is a a subset of R's attributes, the **projection** operator π_L filters the attributes of a relation R.

$$\pi_L(R) \tag{5}$$

Example 1.3 (Projection Operation)

Given $R(A,B,C) = \{(1,2,3),(4,2,3),(4,5,6),(2,5,3),(1,2,6)\}$
Compute $\pi_{C,B}(R)$ and identify one tuple.

1. $(4,2)$
2. $(\mathbf{3},\mathbf{2})$ This is correct.
3. $(4,2,3)$ Projection onto C,B should only have two attributes, not three.
4. $(2,3)$ This reverses the specified projection order of C,B.

1.2.3 Product and Join

Definition 1.10 (Cartesian Product)

The **cartesian product** of two not-necessarily compatible relations R and S is the relation

$$R \times S = \{r \in S, s \in S\} \tag{6}$$

which has a length of $|R| \times |S|$. It is commutative (again, tuples are not ordered despite its name), and if S and R have the same attribute name n, then we usually prefix it by the relation to distinguish it: $S.n, R.n$.

Definition 1.11 (Theta-Join)

The **theta-join** with **join condition/predicate** p gives

$$R \bowtie_p S = \sigma_p(R \times S) \tag{7}$$

If p consists of only equality conditions, then it is called an **equi-join**.

Example 1.4 (Conditional Join Query)

Given $R(A, B) = \{(1, 2), (3, 4), (5, 6)\}$ and
$S(B, C, D) = \{(2, 4, 6), (4, 6, 8), (4, 7, 9)\}$
Compute the join with conditions $R.A < S.C$ AND $R.B < S.D$.

```
SELECT A, R.B, S.B, C, D
FROM R, S
WHERE R.A < S.C AND R.B < S.D
```

1. $(5, 6, 2, 4, 6)$ This doesn't satisfy R.A $<$ S.C as $5 > 4$.
2. $(1, 2, 4, 4, 6)$ The values don't properly satisfy both conditions.
3. $(5, 6, 4, 6, 9)$ Doesn't satisfy conditions as $5 < 6$.
4. $(\mathbf{1}, \mathbf{2}, \mathbf{4}, \mathbf{7}, \mathbf{9})$ Valid result as $1 < 7$ and $2 < 9$, satisfying both conditions.

Definition 1.12 (Natural/Inner Join)

In a θ-join, if p is not specified ($R \bowtie S$), then this is called a **natural join**. The p is automatically implied to be $R.A = S.A$ for all $A \in R.\mathbf{A} \cap S.\mathbf{A}$, and if $R.\mathbf{A} \cap S.\mathbf{A} = \emptyset$, then this is reduced to the cartesian product.

Example 1.5 (Natural Join Operation)

Given $R(A, B) = \{(1, 2), (3, 4), (5, 6)\}$ and
$S(B, C, D) = \{(2, 4, 6), (4, 6, 8), (4, 7, 9)\}$
Find $R \bowtie S$ and identify one resulting tuple.

1. $(1, 2, 6, 8)$ The values don't match on the joining attribute B.
2. $(5, 6, 7, 8)$ Not a valid join result as these values don't align on B.
3. $(1, 2, 4, 8)$ The values don't match properly in the join condition.
4. $(\mathbf{3}, \mathbf{4}, \mathbf{6}, \mathbf{8})$ Valid join result as B=4 matches between R and S, and remaining values align correctly.

Lemma 1.2 (Deletion of Duplicate Attribute in Natural Join)

There is a difference between a natural join and a theta-join with its explicitly written join predicate counterpart. Say that R and S both have attribute A.

1. In theta-join, $R \bowtie_p S$ will contain $R.A$ and $S.A$.
2. In natural join, $R \bowtie S$ will contain A only.

Example 1.6 (Simple Filter)

Find all the addresses of the bars that Ben goes to.

name	address
The Edge	108 Morris Street
Satisfaction	905 W. Main Street

Table 1: Bar Information

drinker	bar	times_a_week
Ben	Satisfaction	2
Dan	The Edge	1
Dan	Satisfaction	2

Table 2: Frequents Information

We do the following.

$$\pi_{\text{address}}\big(\text{Bar} \bowtie_{\text{name=bar}} \sigma_{\text{drinker=Dan}}(\text{Frequents})\big) \tag{8}$$

There is an extension of relational algebra that uses more join operators. We introduce them now. While natural/inner joins result in a relation of matching tuples in the two operands, and outer join contains these tuples and additionally some tuples formed by extending an unmatched tuple in one of the operands.

Definition 1.13 (Left Outer Join)

A **left outer join** $R ⟕ S$ results in the set of all tuples in R, plus those in S that matches (equal in shared attributes) with R. If a r doesn't match with any S, then we fill the S attributes with ω.

$$R ⟕ S = (R \bowtie S) \cup ((R - \pi_{R.\mathbf{A}}(R \bowtie S)) \times \{(\omega, \ldots, \omega)\}) \tag{9}$$

Definition 1.14 (Right Outer Join)

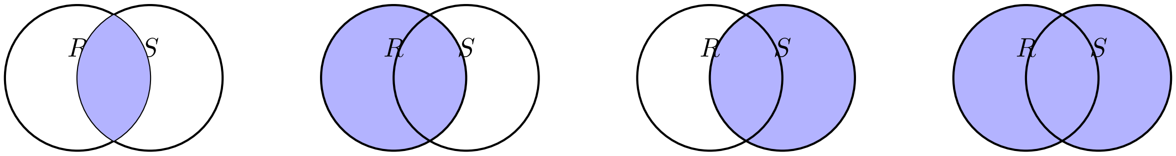
A **right outer join** $R ⟖ S$ results in the set of all tuples in S, plus those in $r \in R$ that matches (equal in shared attributes) with some $s \in S$. If a $s \in S$ doesn't match with any $r \in R$, then we fill the R attributes with ω.

$$R ⟖ S = (R \bowtie S) \cup (\{(\omega, \ldots, \omega)\} \times (R - \pi_{R.\mathbf{A}}(R \bowtie S))) \tag{10}$$

Definition 1.15 (Full Outer Join)

A **full outer join** $R ⟗ S$ results in the set of all tuples where their is a match in either left or right tables. If a $r \in R$ doesn't match with a s, or a s doesn't match with a r, we fill those values in with null.

$$R ⟗ S = (R ⟕ S) \cup (R ⟖ S) \tag{11}$$



(a) Inner Join (b) Left Join (c) Right Join (d) Full Outer Join

Figure 2: Database join operations represented using Venn diagrams.

Example 1.7 (Outer Join)

Consider the following relations.

Name	EmpId	DeptName
Harry	3415	Finance
Sally	2241	Sales
George	3401	Finance
Harriet	2202	Sales
Tim	1123	Executive

DeptName	Manager
Sales	Harriet
Production	Charles

We demonstrate the join operations.

1. Left outer join.

Name	EmpId	DeptName	Manager
Harry	3415	Finance	ω
Sally	2241	Sales	Harriet
George	3401	Finance	ω
Harriet	2202	Sales	Harriet
Tim	1123	Executive	ω

2. Right outer join.

Name	EmpId	DeptName	Manager
Sally	2241	Sales	Harriet
Harriet	2202	Sales	Harriet
ω	ω	Production	Charles

3. Full outer join.

Name	EmpId	DeptName	Manager
Harry	3415	Finance	ω
Sally	2241	Sales	Harriet
George	3401	Finance	ω
Harriet	2202	Sales	Harriet
Tim	1123	Executive	ω
ω	ω	Production	Charles

1.2.4 Renaming

Definition 1.16 (Renaming)

Given a relation R,

1. $\rho_S(R)$ means that you are chaning the relation name to S.
2. $\rho_{(A_1,\ldots,A_n)}(R)$ renames the attribute names to $(A_1, \ldots, A_n)$.
3. $\rho_{S(A_1,\ldots,A_n)}(R)$ renames the relation name to S and the attribute names to $(A_1, \ldots, A_n)$.

It does not really adding any processing power. It is only used for convenience.

1.2.5 Monotonicity

Definition 1.17 (Monotone Operators)

An operator $O(R)$ is monotone with respect to input R if increasing the size (number of rows/tuples) of R does not decrease the output relation O.

$$R \subset R' \implies O(R) \subset O(R') \tag{12}$$

Example 1.8 (Monotone Operators)

Let's go through to see if each operator is monotone.

1. Selection is monotone.
2. Projection is monotone.
3. Cross Product is monotone.
4. Join is monotone.
5. All joins are monotone w.r.t. all parameters.
6. Union is monotone.
7. Intersection is monotone.
8. Difference $R - S$ is monotone w.r.t. R but not monotone w.r.t. S.

Example 1.9 (Getting maximum of an attribute)

Given a schema $R(a, b, c)$, how do we find the maximum of a? This is hard to come up in the first time since we are not allowed to compare across lines. However, we can do the following:

1. Take the cross product of $R_1(a_1, b_1, c_1) \times R_2(a_2, b_2, c_2)$.
2. Select all tuples that are not maxes by selecting all rows where $a_1 < a_2$. The resulting rows will contain only those that are not maximums.

Therefore, we do the following

$$\max_a(R) = [\pi_a(R) \times \pi_a(R)] - \sigma_{a_1 < a_2}[\pi_{a \mapsto a_1}(R) \times \pi_{a \mapsto a_2}(R)] \tag{13}$$

The minimum can be solved analogously.

Notice that the $\max_{att}$ operator is *not* monotone, since the old answer is overwritten.

$$\{\text{oldmax}\} \not\subset \{\text{newmax}\} \tag{14}$$

Generally, whenever we want to construct a non-monotone operator, we want to use the set difference since the composition of monotones is monotone.

You should determine when to project, before or after the difference.

1.3 Constraints

Like mathematical structures, relational databases would not be very useful if they didn't have any structure on them. Our final database property after structure and operations are *constraints*, which can also be written in relational algebra. Through it may not seem obvious, all constraints can be written with what we call *set constraints*.

Definition 1.18 (Set Constraints)

There are two ways in which we can use relational algebra to express constraints. If R and S are relations, then

1. $R = \emptyset$ constrains R to be empty.
2. $R \subset S$ constrains R to be a subset of S.[a]

[a] Note that this is technically unnecessary, since we can write $R - S = \emptyset$. We can also write $R = \emptyset \iff R \subset \emptyset$.

There are three main constraints we will look at:

1. domain constraints
2. key constraints
3. referential integrity

1.3.1 Domain Constraints

There is not much to say here. We have already looked at this constraint when defining the type T of an attribute A. That is, an instance of an attribute A must be of type T: $a \in T$.

Definition 1.19 (Domain Constraints)

We can also constrain the domain of a certain attribute r of relation R. Let $C(r)$ be the constraint. Then,

$$\sigma_{\text{not } C(r)}(R) = \emptyset \tag{15}$$

1.3.2 Key Constraints

Definition 1.20 (Key)

A set of attributes $\mathcal{K} \subset \mathbf{A}$ form a **key** for a relation

1. if we do not allow two tuples in any relation instance to have the same values in *all* attributes of the key (i.e. in general).
2. no proper subset of $\mathcal{K}$ can also be a key for *any* relation instance, that is, $\mathcal{K}$ is *minimal.*[a]

A relation may have multiple keys, but we typically pick one as the **primary key** and underline all its attributes in the schema, e.g. `Address(street, city, state, zip)`.

[a] By minimal we do not mean that the number of attributes in K is minimal. It is minimal in the sense that no proper subset of $\mathcal{K}$ can be a key.

While we can make a key with a set of attributes, many databases use artificial keys such as unique ID numbers for safety.

Example 1.10 (Keys of User Relation)

Given the schema `User(uid, name, age)`,

1. `uid` is a key of `User`
2. `age` is not a key (not an identifier) even if the relation at the current moment all have different ages.
3. {`uid, name`} is not a key (not minimal)

Definition 1.21 (Key Constraints)

If we have the key $\mathcal{K} = (k_1, \ldots, k_m) \subset \mathbf{A}$ of a relation R, we can express this constraint as such. We rename R to copies R_1, R_2, denote $\mathcal{K}' = \mathbf{A} - \mathcal{K}$, and write

$$\sigma_{R_1.\mathcal{K}=R_2.\mathcal{K} \text{ and } R_1.\mathcal{K}' \neq R_2.\mathcal{K}'}(R_1 \times R_2) = \emptyset \tag{16}$$

That is, if we found a $r_1 \in R_1, r_2 \in R_2$ that matched in the key attributes, then it must be the case that $r_1 = r_2$. But if they are equal in the rest of the attributes, they will be projected out.

1.3.3 Referential Integrity

Definition 1.22 (Referential Integrity Constraints)

One way that we can use this is through *referential integrity* constraints, which asserts that a value appearing as an attribute r in relation R also should appear in a value of an attribute s in relation S. That is,

$$\pi_r(R) \subset \pi_s(S) \tag{17}$$

1.4 Functional Dependencies

A generalization of keys is functional dependencies. Since it can be viewed as another form of key constraint, it is included here, but it is also a method of describing relationships within data, which will be useful in effective designing of databases.

Definition 1.23 (Functional Dependency)

Given a relation R with attributes $\mathbf{A}$, let $\mathbf{a} = (a_1, \ldots, a_n), \mathbf{b} = (b_1, \ldots, b_m) \subset \mathbf{A}$. Then, the constraint

$$\mathbf{a} \mapsto \mathbf{b} \tag{18}$$

also called "**a functionally determines b**," means that if two tuples agree on **a**, then they must agree on **b**. We say that R satisfies a FD $f : \mathbf{a} \mapsto \mathbf{b}$ or a set of FDs $F = \{f\}$ if this constraint is satisfied.

From this, we can see that the term "functional" comes from a literal function being defined on the input **a**.

Lemma 1.3 (FDs as Key Constraints)

Note that the functional dependency $\mathbf{a} \mapsto \mathbf{b}$ also implies the key constraint

$$\sigma_{R_1.\mathbf{a}=R_2.\mathbf{a} \text{ and } R_1.\mathbf{b} \neq R_2.\mathbf{b}'}(R_1 \times R_2) = \emptyset \tag{19}$$

Definition 1.24 (Superkey)

A set of attributes **k** of a relation R is called a **superkey** if

$$\mathbf{k} \mapsto \mathbf{r} - \mathbf{k} \tag{20}$$

If no $\mathbf{k}' \subset \mathbf{k}$ functionally determines **r**, then it is a key.

Example 1.11 (Clarification of Minimal Meaning)

Given a relation $R(A, B, C, D, E, F)$ with functional dependencies

$$AEF \mapsto C, BF \mapsto C, EF \mapsto D, ACDE \mapsto F \tag{21}$$

We can see that every attribute not on the right hand side (C, D, F) must be a key. If we do a bit of testing out, we can see that

1. $ABEF$ is a key.
2. $ABCDE$ is also a key since even though it has 5 attributes, it is minimal in the sense that no proper subset functionally determines every other attribute.

1.4.1 Structure on Spaces of Functional Dependencies

To introduce additional structure, we will introduce two spaces.

1. Given a relation R, let us consider the set of all FDs $F = F(R)$ on R. This is clearly a large set, which increases exponentially w.r.t. the number of attributes in R.
2. Let us denote the set of all relations R satisfying F as R_F, which is an infinite set.

Theorem 1.4 (Armstrong Axioms)

Let's prove a few properties of FDs, which have nice structure.

1. *Splitting and Combining.* The two sets of FDs are equal.

$$\{\mathbf{a} \mapsto \mathbf{b}\} \iff \{\mathbf{a} \mapsto b_i \mid i = 1, \ldots, m\} \tag{22}$$

2. *Trivial FDs.* Clearly elements of **a** uniquely determines its own attributes.

$$\mathbf{a} \mapsto \mathbf{b} \implies \mathbf{a} \mapsto \mathbf{b} - \mathbf{a} \tag{23}$$

or can also be written as

$$\mathbf{b} \subset \mathbf{a} \implies \mathbf{a} \mapsto \mathbf{b} \tag{24}$$

3. *Augmentation.*

$$\mathbf{a} \mapsto \mathbf{b} \implies \mathbf{a}, \mathbf{c} \mapsto \mathbf{b}, \mathbf{c} \tag{25}$$

4. *Transitivity.* If $\mathbf{a} \mapsto \mathbf{b}, \mathbf{b} \mapsto \mathbf{c}$, then

$$\mathbf{a} \mapsto \mathbf{c} \tag{26}$$

Proof. Trivial.

It is also possible to put a partial order on F.

Definition 1.25 (Partial Order)

Given two FDs f and g, consider the set of all relations R satisfying f and g, denoted as R_f and R_g.

1. Then $f \implies g$ iff $R_f \subset R_g$.
2. $f \iff g$ iff $R_f = R_g$.

Moreover, we can use this structure on F to induce structure on the set of attributes **r**.

Definition 1.26 (Closure of Attributes)

The **closure** of $\mathbf{r}$ under a set of FDs F is the set of attributes $\mathbf{b}$ s.t.

$$R_F = R_{\mathbf{b}} \tag{27}$$

We denote this as $\mathbf{b} = \mathbf{r}^+$. To actually compute the closure, we take a greedy approach by starting with $\mathbf{r}$ and incrementally adding attributes satisfying F until we cannot add any more.

Theorem 1.5 (Implication of Functional Dependencies)

If we want to know where one FD $f : \mathbf{a} \mapsto \mathbf{b}$ follows from a set F of functional dependencies,

1. We compute the closure $\mathbf{a}^+$ w.r.t. F.
2. If $\mathbf{b} \subset \mathbf{a}^+$, then f follows from F.

Alternatively, we can also use the *Armstrong axioms* above to derive all implications.

1.4.2 Projections of Functional Dependencies

If we have a relation R with a set of FDs F, and we project $R' = \pi_{\mathbf{r}'}(R)$, then the set of FDs F' that hold for R' consists of

1. The FDs that follow from F, and
2. involve only attributes of R.

Calculating the FDs that hold is not trivial, and the calculations of the FDs for R' is exponential in the number of attributes of R'.

Theorem 1.6 (Calculating Projections of Functional Dependencies)

Given a relation R and its set of functional dependencies F, let $R' = \pi(R)$ be a projection.

1. Let G be the set of FDs that we construct that hold for R'. Initially it is empty.
2. For each set of attributes $\mathbf{r}$ that is a subset of the attributes of R_1, compute $\mathbf{r}^+$. This computation is performed w.r.t. the set of FDs in S, and may involve attributes in the schema of R but not in R'. Add to G all nontrivial FDs $\mathbf{r} \to A$ such that A is both in $\mathbf{r}^+$ and an attribute in R'.

This gives our projection, but it may not be minimal. To make it minimal, repeat the two steps, independently.

1. If there is an FD g in G that follows from the other FDs in G, remove g.
2. Let $g = \mathbf{a} \mapsto \mathbf{b}$ be an FD in G, with at least 2 attributes in $\mathbf{a}$, and let $\mathbf{a}'$ be $\mathbf{a}$ with one of its attributes removed. If $g' = \mathbf{a}' \mapsto \mathbf{b}$ follows from the FDs in G (including g), then replace g with g'.

Once neither step can be done, we have a minimal set.

Example 1.12 (Computing FDs for Projections)

Suppose $R(A, B, C, D)$ has FD's $A \to B$, $B \to C$, and $C \to D$. Suppose also that we wish to project out the attribute B, leaving a relation $R_1(A, C, D)$. In principle, to find the FD's for R_1, we need to take the closure of all eight subsets of $\{A, C, D\}$, using the full set of FD's, including those involving B. However, there are some obvious simplifications we can make.

- Closing the empty set and the set of all attributes cannot yield a nontrivial FD.
- If we already know that the closure of some set X is all attributes, then we cannot discover any new FD's by closing supersets of X.

Thus, we may start with the closures of the singleton sets, and then move on to the doubleton sets if

necessary. For each closure of a set X, we add the FD $X \to E$ for each attribute E that is in X^+ and in the schema of R_1, but not in X.
First, $\{A\}^+ = \{A, B, C, D\}$. Thus, $A \to C$ and $A \to D$ hold in R_1. Note that $A \to B$ is true in R, but makes no sense in R_1, because B is not an attribute of R_1.
Next, we consider $\{C\}^+ = \{C, D\}$, from which we get the additional FD $C \to D$ for R_1. Since $\{D\}^+ = \{D\}$, we can add no more FD's, and are done with the singletons.
Since $\{A\}^+$ includes all attributes of R_1, there is no point in considering any superset of $\{A\}$. The reason is that whatever FD we could discover, for instance $AC \to D$, follows from an FD with only A on the left side: $A \to D$ in this case. Thus, the only doubleton whose closure we need to take is $\{C, D\}^+ = \{C, D\}$. This observation allows us to add nothing. We are done with the closures, and the FD's we have discovered are $A \to C$, $A \to D$, and $C \to D$.
If we wish, we can observe that $A \to D$ follows from the other two by transitivity. Therefore a simpler, equivalent set of FD's for R_1 is $A \to C$ and $C \to D$. This set is, in fact, a minimal basis for the FD's of R_1.

2 Design Theory for Relational Databases

Okay, we know all that is needed to define a database, but not how to work practically well with it yet. Given some set of data or some structure, we must find an optimal way to store it, within either one or multiple relations. How should we do this? The first step is to categorize the potential problems.

2.1 Anomalies and Decomposition

Definition 2.1 (Anomaly)

Beginners often try to cram too much into a relation, resulting in **anomalies** of three forms.

1. *Redundancies.* Information repeated unnecessarily in several tuples.
2. *Updates.* Updating information in one tuple can leave the same information unchanged in another (which violates referential integrity).
3. *Deletion.* If a set of values becomes empty, we may lose other information as a side effect (another violation of referential integrity).

Example 2.1 (Redundancies)

Let's try to see where these anomalies came from. Consider a non-trivial FD $\mathbf{a} \mapsto \mathbf{b}$ where $\mathbf{a}$ is not a superkey. Since $\mathbf{a}$ is not a superkey, there are attributes, say $\mathbf{c}$ that are not functionally determined by $\mathbf{a}$. Therefore, there are multiple combinations of $\mathbf{a}, \mathbf{b}, \mathbf{c}$ which have the same $\mathbf{a}, \mathbf{b}$ but not $\mathbf{c}$, leading to redundancy.

a	**b**	**c**
x	y	z_1
$\vdots$	$\vdots$	$\vdots$
x	y	z_{100}

Table 3: Redundant information in **a** and **b** attributes.

If $T_{\mathbf{a}}, T_{\mathbf{b}}$ are both 4-byte integers, we have just wasted 400 bytes of space. It seems that most of the redundancy comes from using a and b too many times. If we had two relations R and S.

To eliminate these anomalies, we want to **decompose** relations, which involve splitting R into two new relations R_1, R_2.

Definition 2.2 (Decomposition)

Given relation with schema $R(\mathbf{A})$, we can decompose R into two relations $R_1(\mathbf{A}_1)$ and $R_2(\mathbf{A}_2)$ such that

1. $\mathbf{A} = \mathbf{A}_1 \cup \mathbf{A}_2$ (not necessarily disjoint)
2. $R_1 = \pi_{\mathbf{A}_1}(R)$
3. $R_2 = \pi_{\mathbf{A}_2}(R)$

There are two types of decomposition:

1. **lossy** decomposition of R to R_1, R_2 means that joining R_1, R_2 does not give us R.
2. **lossless** decomposition indeed gives us back R.

Theorem 2.1 (Decomposition)

Any decomposition R_1, R_2 of R satisfies

$$R \subset R_1 \bowtie R_2 \tag{28}$$

Proof. If $R_1.\mathbf{A} \cap R_2.\mathbf{A} = \emptyset$, then $R \subset R_1 \times R_2$ for sure. If there is some overlap, then given some $r \in R$, we will be able to find $r_1 \in R_1$ and $r_2 \in R_2$, and they will definitely join in their overlapping attribute, giving R.

Therefore, we should be worried about if our decomposition will create *new* tuples since it will never delete relevant ones.

Example 2.2 (Lossy and Lossless Decomposition)

Consider the relation

X	**Y**	**Z**
a	b	c_1
a	b	c_2
a_1	b	c_2

Projecting to (X, Y) and (X, Z) gives us a lossless decomposition.

X	**Y**
a	b
a_1	b

X	**Z**
a	c_1
a	c_2
a_1	c_2

X	**Y**	**Z**
a	b	c_1
a	b	c_2
a_1	b	c_2

While projecting to (X, Y) and (Y, Z) gives us a lossy one since we get (a_1, b, c_1) when joining the decompositions, which is not in the original relation.

X	**Y**
a	b
a_1	b

Y	**Z**
b	c_1
b	c_2

X	**Y**	**Z**
a	b	c_1
a	b	c_2
a_1	b	c_1
a_1	b	c_2

Generally, the intuition behind trying to find a lossless decomposition is this: For a value of a certain attribute X that is in both decompositions R_1 and R_2, we want only one instance of a value on one side. For example,

1. in the lossless decomposition, notice how for overlapping attribute $\mathbf{X}$, for each instance a, a_1, we had 1 of each on the left relation, and however many on the right.
2. in the lossy decomposition, notice how we have two b's on the left and two b's on the right for the lossy decomposition. Rather, we want something like 1 b vs multiple b's.

2.1.1 Boyce-Codd Normal Form

This decomposition eliminated the redundancy anomaly, and for attributes **X** and **Z**, it eliminated the deletion and update anomalies. Let's formalize the conditions needed to decompose such a relation, and how we should actually decompose it.

Definition 2.3 (BNCF)

BCNF is defined in two equivalent ways:

1. A relation R is in **BNCF** iff whenever there is a nontrivial FD $\mathbf{a} \mapsto \mathbf{b}$, it is the case that $\mathbf{a}$ is a superkey for R.
2. Given a relation R with a set of nontrivial functional dependencies $F = \{\mathbf{a} \mapsto \mathbf{b}\}$, it is in BCNF if for every $\mathbf{a}$, $\mathbf{a}^+$ is the set of all attributes of R.

Note that since there are multiple keys, $\mathbf{a}$ does not always have to include the same key.

Example 2.3 (Non-BNCF Form)

The table below is not in BCNF form since

$$(\texttt{title}, \texttt{year}) \mapsto (\texttt{length}, \texttt{genre}, \texttt{studioName}) \tag{29}$$

Is a functional dependency where the LHS is not a superkey (the key is `title`, `year`).

title	**year**	**length**	**genre**	**studioName**	**starName**
Star Wars	1977	124	SciFi	Fox	Carrie Fisher
Star Wars	1977	124	SciFi	Fox	Mark Hamill
Star Wars	1977	124	SciFi	Fox	Harrison Ford
Gone With the Wind	1939	231	drama	MGM	Vivien Leigh
Wayne's World	1992	95	comedy	Paramount	Dana Carvey
Wayne's World	1992	95	comedy	Paramount	Mike Meyers

Table 4: Movie Data

Example 2.4 (BNCF Form)

However, if we decompose this into the following tables, both satisfy BCNF.

title	**year**	**length**	**genre**	**studioName**
Star Wars	1977	124	sciFi	Fox
Gone With the Wind	1939	231	drama	MGM
Wayne's World	1992	95	comedy	Paramount

Table 5: Simplified Movie Data

title	**year**	**starName**
Star Wars	1977	Carrie Fisher
Star Wars	1977	Mark Hamill
Star Wars	1977	Harrison Ford
Gone With the Wind	1939	Vivien Leigh
Wayne's World	1992	Dana Carvey
Wayne's World	1992	Mike Meyers

Table 6: Movie Titles, Years, and Stars

Algorithm 2.0 (Constructing BCNF of a Relation)

To actually construct this, we act on BCNF violations. Given that we have found a FD $\mathbf{a} \mapsto \mathbf{b}$ that doesn't satisfy BCNF (i.e. $\mathbf{a}$ is not a superkey) of relation R, we decompose it into the following R_1 and R_2.

1. We want $\mathbf{a}$ to be a superkey for one of the subrelations, say R_1. Therefore, we have it satisfy $\mathbf{a} \mapsto \mathbf{a}^+$, which is satisfied by definition, and set
$$R_1 = \pi_{\mathbf{a},\mathbf{a}^+}(R) \tag{30}$$
2. We don't want any loss in data, so we take the rest of the attributes not in the closure and define
$$R_2 = \pi_{\mathbf{a},\mathbf{r}-\mathbf{a}^+}(R) \tag{31}$$

We keep doing this until every subrelation satisfies BCNF. This is guaranteed to terminate since we are decreasing the size of the relations until all attributes are superkeys.

Theorem 2.2 (Lossless Guarantee for BCNF)

If we decompose on a BCNF violation as stated above, then our decomposition is guaranteed to be a lossless join decomposition.

Proof. An outline of the proof means that given a BCNF violation $\mathbf{a} \mapsto \mathbf{b}$, we must show that anything we project always comes back in the join
$$R \subset \pi_{\mathbf{a}^+}(R) \bowtie \pi_{\mathbf{a},\mathbf{r}-\mathbf{a}}(R) \tag{32}$$
which is proved trivially, and anything that comes back in the join must be in the original relation
$$\pi_{\mathbf{a}^+}(R) \bowtie \pi_{\mathbf{a},\mathbf{r}-\mathbf{a}}(R) \subset R \tag{33}$$
We can use the fact that $\mathbf{a} \mapsto \mathbf{b}$.

The lossless guarantee makes BCNF very attractive, but it comes with a tradeoff.

Theorem 2.3 (BCNF Does Not Preserve Functional Dependencies)

BCNF may not enforce equivalent functional dependencies over its decomposed relations compared to its original relation.

Proof. We consider a counterexample. Given a schema $R(A, B, C)$ with FDs
$$f = AB \mapsto C, \;\; g = C \mapsto B \tag{34}$$
When we decompose based on $C \mapsto B$ into $R_1(C, B), R_2(C, A)$, the join is still lossless because we kept C in both relations, and when we join the two on C, each C value will match exactly one B value (due to $C \to B$ in R_2). However, we cannot enforce $AB \to C$.

It is initially confusing since one may ask that if functional dependencies aren't preserved, then doesn't this mean lossy decomposition? Not exactly, since the lossless join property is independent of dependency preservation. The lossless join property ensures we can reconstruct the exact original relation through natural joins, without getting any spurious tuples. Dependency preservation simply means we can't enforce a FD when inserting/updating. In fact, this is analogous to some functional dependencies being removed when projecting a relation.

2.1.2 Recovery and Chase Test

Now let's talk about properties of general decompositions.

Theorem 2.4 (Any 2-Attribute Relation Satisfies BNCF)

Any 2-attribute relations is in BCNF. Let's label the attributes a, b and go through the cases.

1. There are no nontrivial FDs, meaning that $\{A, B\}$ is the only key. Then BCNF must hold since only a nontrivial FD can violate this condition.
2. $a \mapsto b$ holds but not $b \mapsto a$, meaning that a is the only key. Thus there is no violation since a is a superkey.
3. $b \mapsto a$ holds but not $a \mapsto b$. This is symmetric as before.
4. Both hold, meaning that both a and b are keys. Since any FD has at least one of a, b on the left, this is satisfied.[a]

[a] Note that BCNF only requires *some* key to be contained on the left side, not that all keys are.

From this theorem stating that any 2-attribute relation satisfies BCNF, we can just trivially decompose all relations into relations $R_1, \ldots, R_n$, and we are done, right? Not exactly, since to ensure lossless join, we must start with the original relation and *split only on the BCNF violations*. This is what makes it lossless, and simply splitting arbitrarily will lead to a lossy decomposition. Furthermore, even if it was lossless, it will destroy most functional dependencies too.

This motivates the need for a general test to see if a decomposition of R is lossless. That is, is it true that $\pi_{S_1}(R) \bowtie \pi_{S_2}(R) \bowtie \cdots \bowtie \pi_{S_k}(R) = R$? Three important things to remember are:

- The natural join is associative and commutative. It does not matter in what order we join the projections; we shall get the same relation as a result. In particular, the result is the set of tuples t such that for all $i = 1, 2, \ldots, k$, t projected onto the set of attributes S_i is a tuple in $\pi_{S_i}(R)$.
- Any tuple t in R is surely in $\pi_{S_1}(R) \bowtie \pi_{S_2}(R) \bowtie \cdots \bowtie \pi_{S_k}(R)$. The reason is that the projection of t onto S_i is surely in $\pi_{S_i}(R)$ for each i, and therefore by our first point above, t is in the result of the join.
- As a consequence, $\pi_{S_1}(R) \bowtie \pi_{S_2}(R) \bowtie \cdots \bowtie \pi_{S_k}(R) = R$ when the FD's in F hold for R if and only if every tuple in the join is also in R. That is, the membership test is all we need to verify that the decomposition has a lossless join.

Theorem 2.5 (Chase Test for Lossless Join)

The **chase test** for a lossless join is just an organized way to see whether a tuple t in $\pi_{S_1}(R) \bowtie \pi_{S_2}(R) \bowtie \cdots \bowtie \pi_{S_k}(R)$ can be proved, using the FD's in F, also to be a tuple in R. If t is in the join, then there must be tuples in R, say $t_1, t_2, \ldots, t_k$, such that t is the join of the projections of each t_i onto the set of attributes S_i, for $i = 1, 2, \ldots, k$. We therefore know that t_i agrees with t on the attributes of S_i, but t_i has unknown values in its components not in S_i.

We draw a picture of what we know, called a *tableau*. Assuming R has attributes $A, B, \ldots$ we use $a, b, \ldots$ for the components of t. For t_i, we use the same letter as t in the components that are in S_i, but we subscript the letter with i if the component is not in S_i. In that way, t_i will agree with t for the attributes of S_i, but have a unique value — one that can appear nowhere else in the tableau — for other attributes.

Example 2.5

Suppose we have relation $R(A, B, C, D)$, which we have decomposed into relations with sets of attributes $S_1 = \{A, D\}$, $S_2 = \{A, C\}$, and $S_3 = \{B, C, D\}$. Then the tableau for this decomposition is:

A	B	C	D
a	b_1	c_1	d
a	b_2	c	d_2
a_3	b	c	d

The first row corresponds to set of attributes A and D. Notice that the components for attributes A and D are the unsubscripted letters a and d. However, for the other attributes, b and c, we add the subscript 1 to indicate that they are arbitrary values. This choice makes sense, since the tuple (a, b_1, c_1, d) represents a tuple of R that contributes to $t = (a, b, c, d)$ by being projected onto $\{A, D\}$ and then joined with other tuples. Since the B- and C-components of this tuple are projected out, we know nothing yet about what values the tuple had for those attributes.
Similarly, the second row has the unsubscripted letters in attributes A and C, while the subscript 2 is used for the other attributes. The last row has the unsubscripted letters in components for $\{B, C, D\}$ and subscript 3 on a. Since each row uses its own number as a subscript, the only symbols that can appear more than once are the unsubscripted letters.

Remember that our goal is to use the given set of FD's F to prove that t is really in R. In order to do so, we "chase" the tableau by applying the FD's in F to equate symbols in the tableau whenever we can. If we discover that one of the rows is actually the same as t (that is, the row becomes all unsubscripted symbols), then we have proved that any tuple t in the join of the projections was actually a tuple of R.

To avoid confusion, when equating two symbols, if one of them is unsubscripted, make the other be the same. However, if we equate two symbols, both with their own subscript, then you can change either to be the other. However, remember that when equating symbols, you must change all occurrences of one to be the other, not just some of the occurrences.

Example 2.6

Let us continue with the decomposition of the previous example, and suppose the given FD's are $A \to B$, $B \to C$, and $CD \to A$. Start with the tableau:

A	B	C	D
a	b_1	c_1	d
a	b_1	c	d_2
a_3	b	c	d

Since the first two rows agree in their A-components, the FD $A \to B$ tells us they must also agree in their B-components. That is, $b_1 = b_2$. We can replace either one with the other, since they are both subscripted. Let us replace b_2 by b_1. Then after applying $B \to C$:

A	B	C	D
a	b_1	c	d
a	b_1	c	d_2
a_3	b	c	d

Next, we observe that the first and third rows agree in both columns C and D. Thus, we may apply the FD $CD \to A$ to deduce that these rows also have the same A-value; that is, $a = a_3$. We replace a_3 by a, giving us:

A	B	C	D
a	b_1	c	d
a	b_1	c	d_2
a	b	c	d

At this point, we see that the last row has become equal to t, that is, (a, b, c, d). We have proved that if R satisfies the FD's $A \to B$, $B \to C$, and $CD \to A$, then whenever we project onto $\{A, D\}$, $\{A, C\}$, and $\{B, C, D\}$ and rejoin, what we get must have been in R. In particular, what we get is the same as the tuple of R that we projected onto $\{B, C, D\}$.

Example 2.7 (Chase Test for Lossless-join Decomposition)

Consider relation $R(A, B, C, D)$ with functional dependencies:

$$A \to B, \quad C \to D, \quad AD \to C, \quad BC \to A$$

We will perform the chase test on decomposition $\{AB, ACD, BC, BD\}$ to prove it is lossless.

1. Initial chase table with subscript 'a' for known values and variables for unknown:

A	B	C	D	Source
a	b	c	d	Original
a	b	b_1	b_2	From AB
b_3	b_4	c	d	From ACD
b_5	b_4	c	b_6	From BC
b_7	b	b_8	d	From BD

2. Apply functional dependencies sequentially:
   - Apply $A \to B$:
     - Where A values match, B values must match
     - Rows 1 and 2: since $A = a$, B must be same
     - No change needed as $B = b$ in both
   - Apply $C \to D$:
     - Where C values match, D values must match
     - Rows 1, 3, 4: $C = c$, so D must be same
     - b_6 becomes d

     Updated table after $C \to D$:

A	B	C	D
a	b	c	d
a	b	b_1	b_2
b_3	b_4	c	d
b_5	b_4	c	d
b_7	b	b_8	d

   - Apply $AD \to C$:
     - Where A and D match, C must match
     - Row 1 and 2: $A = a$ but different D, no change
   - Apply $BC \to A$:
     - Where B and C match, A must match
     - Row 3 and 4: same B (b_4) and C (c), so A must match
     - b_3 becomes b_5

3. Final chase table:

A	B	C	D
a	b	c	d
a	b	b_1	b_2
b_5	b_4	c	d
b_5	b_4	c	d
b_7	b	b_8	d

4. Conclusion:
   - The chase process shows variable equating ($b_3 = b_5$)
   - The decomposition is lossless because:
     - AB and BC share B, enforcing $BC \to A$
     - ACD preserves both $C \to D$ and $AD \to C$
     - Common attributes ensure lossless reconstruction
     - Each subrelation preserves relevant FDs
   - The natural join: $(AB \bowtie BC) \bowtie ACD \bowtie BD$ recovers R without loss

Therefore, $\{AB, ACD, BC, BD\}$ is a valid lossless-join decomposition in BCNF.

2.2 The Entity-Relationship Model

We have now learned to decompose relations effectively without anomalies, but it is not clear what these relations may represent. When designing a database, it isn't really ideal to just dump all data into a relation and then try to iteratively decompose it until you get a bunch of relations in BCNF. It's better to have a starting plan of what relations you should need and try and construct interpretable relationships between a pair or set of relations. To do this, *Entity-Relationship diagrams* are of great help.

Let's briefly talk about keys again. The most obvious application of keys is allowing lookup of a row by its key value. A more practical application of keys are its way to link key IDs for one relation to another key ID of a different relation. For example, we may have two schemas `Member(uid, gid)` and `Group(gid)`, and we can join these two using the condition `Member.gid = Group.gid`.

Definition 2.4 (Entity-Relationship Model)

We can think of every relation as modeling some *data* or a *relationships between data.*[a] This is shown in an **E/R diagram**.

1. An **entity set** is a relation that contains data, represented as a *rectangle.* Its tuples are called *entities.*[b]
2. A **relationship set** is a relation that contains relationships (sometimes stored as the key pairs of two entity sets), represented as a *diamond.* Its tuples are called **relationships**.[c]
3. **Attributes** are properties of entities or relationships, like attributes of tuples or objects, represented as *ovals.* Key attributes are underlined.

Don't confuse entity sets with entities and relationship sets with relationships.

[a] Sometimes, the line may be blurred, but the designer will have to make the choice.

[b] This is analogous to an entity set being a class and entities as objects.

[c] A minor detail is that relationships aren't really relations since the tuples in relations connect two entities, rather than the keys themselves, so some care must be taken to convert the entities into a set of attributes.

Example 2.8 (E/R Diagram)

Let us model a social media database with the relations

1. `Users(uid, name, age, popularity)` recording information of a user.
2. `Member(uid, gid, from_date)` recording whether a user is in a group and when they first joined.
3. `Groups(gid, name)` recording information of group.

The ER diagram is shown below, where we can see that the Member relation shows a relationship between the two entities Users and Groups.

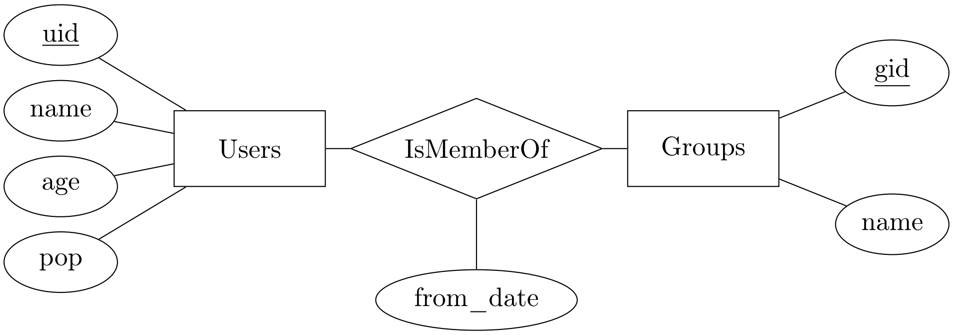


Figure 3: Social media database ER diagram.

Note that the `from_date` attribute must be a part of the Member relation since it isn't uniquely associated with a user (a user can join multiple groups on different dates) or a group (two users can join a group on different dates). If there is an instance that someone joins, leaves, and rejoins a group, then we can modify our design by either:

1. overwriting the first date joined
2. making another relation `MembershipRecords` which has a date also part of the key, which will capture historical membership.

Therefore, we must determine if a relation models an entity or a relationship. There could also be multiple relationship sets between the same entity sets, e.g. if `Member` and `Likes` associates between `Users` and `Groups`.

2.3 Relationships and Multiplicity

A relationship really stores minimal information in the sense that if you have you know the entities it connects, that determines everything about the relation.

Theorem 2.6 (Relationships are Functionally Dependent on Connecting Entities)

In a relationship set, each relationship is uniquely identified by the entities it connects. More formally, if a relationship R connects entities $e_1 \in E_1, \ldots, e_n \in E_n$ with keys $\mathbf{k} = (k_1, \ldots, k_n)$, then $\mathbf{k}$ is a superkey of R.

2.3.1 Multiplicity of Binary Relationships

The 4 categories representing the multiplicity just further categorizes how this functional dependency is realized.

Definition 2.5 (Multiplicity of 2-Way Relationships)

Given that E and F are entity sets,

1. *Many-many*: Each entity in E is related to 0 or more entities in F and vice versa. There are no restrictions, and we have `IsMemberOf(uid, gid)`.[a]

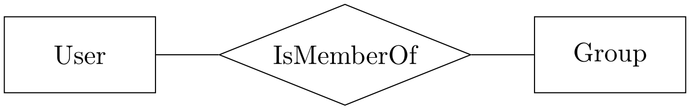
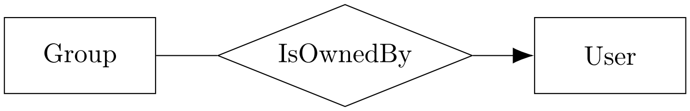
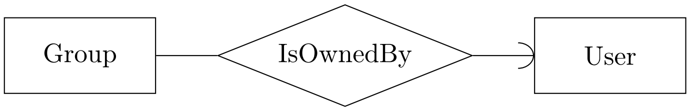


Figure 4

2. *Many-One*: Each entity in E is related to 0 or 1 entities in F, but each entity in F is related to 0 or more in E. If E points to F with a straight arrow, then you can just think that this is a function, and we have `IsOwnedBy(gid, uid)`. If we have a rounded arrow, this means that for each group, its owner `must` exist in `Users` (so no 0).





3. *One-One*: Each entity in E is related to 0 or 1 entity in F and vice versa. We can have either `IsLinkedTo(uid, twitter_uid)` or `IsLinkedTo(uid, twitter_uid)` and must choose a primary key from these two possible keys.

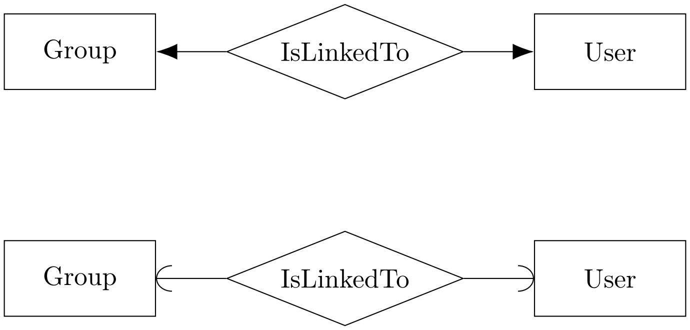


Figure 5: Bidirectional relationship between Group and User entities

[a] Note that this is just a restating of the theorem before.

Theorem 2.7 (Multiplicity with Functional Dependencies)

You may notice that multiplicity and functional dependence are very similar that is. If we have two relations R, S and have a relationship pointing from R to S, then this states the FD $\mathbf{r} \mapsto \mathbf{s}$! Say that the keys are $\mathbf{k}_R, \mathbf{k}_S$, respectively. Then, we have

$$\mathbf{k}_R \mapsto \mathbf{r} \mapsto \mathbf{s} \mapsto \mathbf{k}_S \tag{35}$$

Example 2.9 (Movie Stars)

Given the relations

1. `Movies(title, year, length, name)`
2. `Stars(name, address)` of a movie star and their address.
3. `Studios(name, address)`
4. `StarsIn(star_name, movie_name, movie_year)`
5. `Owns(studio_name, movie_name, movie_year)`

We have the following ER diagram

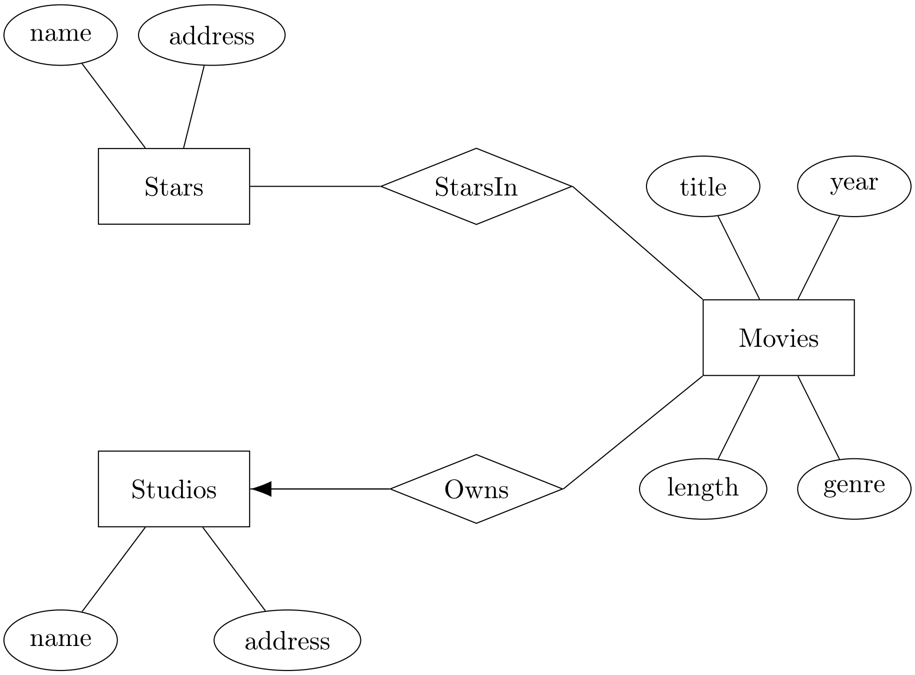


Figure 6: ER diagram showing relationships between Stars, Movies, and Studios entities

Example 2.10 (Relationship within Itself)

Sometimes, there is a relationship of an entity set with itself. This gives the relations

1. `Users(uid, ...)`
2. `IsFriendOf(uid1, uid2)`
3. `IsChildOf(child_uid, parent_uid)`

This can be modeled by the following. Note that

1. users have no limitations on who is their friend.
2. assuming that all parents are single, a person can have at most one parent, so we have an arrow.

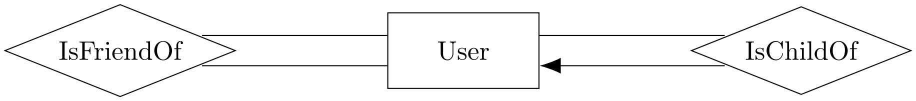


Figure 7: Self-referential relationships for User entity

2.3.2 Multiplicity of Multiway Relationships

Sometimes, it is necessary to have a relationship between 3 or more entity sets. It can be confusing to contruct the relations with the necessary keys. A general rule of thumb for constructing the relation of a relationship is

1. Everything that the arrows point into are not keys.
2. Everything else are keys. So the arrow stumps are keys.

Example 2.11 (Movie Stars)

Suppose that we wanted to model *Contract* relationship involving a studio, a star, and a movie. This relationships represents that a studio had contracted with a particular star to act in a particular movie. We want a contract to be owned by one studio, but one studio can have multiple contracts for different combinations of stars and movies. This gives the relations

1. `Stars(name, address)`
2. `Movies(title, year, length, name)`
3. `Studios(name, address)`
4. `Contracts(star_name, movie_name, studio_name)`

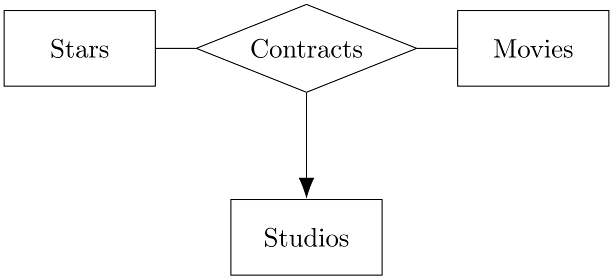


Figure 8

We can make this even more complex by modifying contracts to have a studio of the star and the producing studio.

1. `Contracts(star_name, movie_name, produce_studio_name, star_studio_name)`

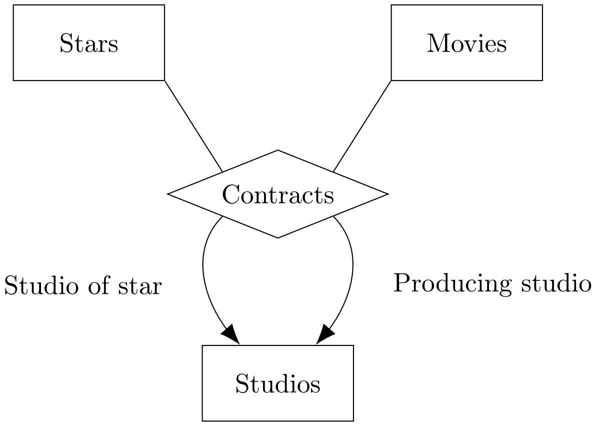


Figure 9: Note that contracts can also have attributes, e.g. salary or time period.

Example 2.12 (Social Media)

In a 3-ary relationship a user must have an initiator in order to join a group. In here, the `isMemberOf` relation has an initiator, which must be unique for each initiated member, for a given group.

1. `User(uid, ...)`
2. `Group(gid, ...)`
3. `IsMemberOf(member, initiator, gid)` since a member must have a unique pair of initiator/-group that they are in.

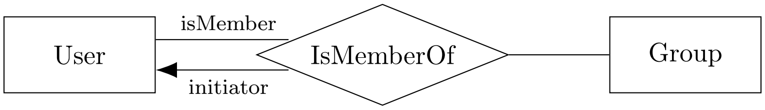


Figure 10: User and Group membership relationship with roles

But can we model n-ary relationships with only binary relationships? Our intuition says we can't, for the same reasons that we get lossy decomposition into 2-attribute schemas when we try to satisfy BCNF.

Example 2.13 (N-ary Relationships vs Multiple Binary Relationships)

N-ary relationships in general cannot be decomposed into multiple binary relationships. Consider the following diagram.

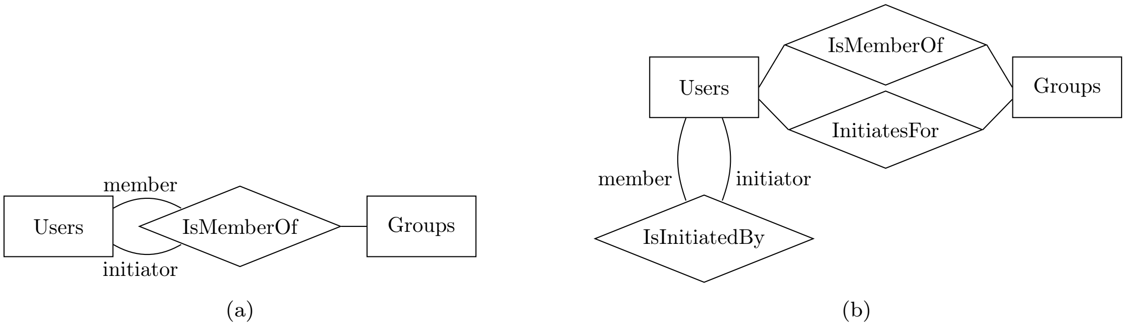


Figure 11: Attempt at reducing nary to binary ER relationships. Are they equivalent?

1. u1 is in both g1 and g2, so `IsMemberOf` contains both (u1, g1) and (u2, g2)
2. u2 served as both an initiator in both g1 and g2, so `InitiatesFor` contains both (g1, u2) and (g2, u2).
3. But in reality, u1 was initiated by u2 for g1 but not u2 for g2. This contradicts the information that you would get when joining the `IsMemberOf` and `InitiatesFor` relations.

Therefore, combining binary relations may generate something spurious that isn't included in the n-ary relationship.

2.4 Subclasses of Entity Sets

Sometimes, an entity set contains certain entities that have special properties not associated with all members of the set. We model this by using a **isa** relationship with a triangle.

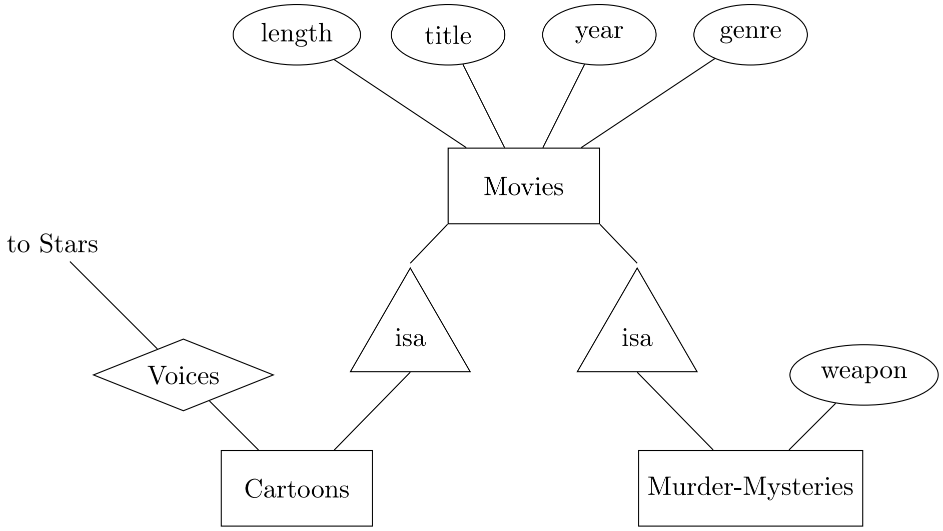


Figure 12: There are two types of movies: cartoons and murder-mysteries, which can have their own sub-attributes and their own relationships.

Suppose we have a tree of entity sets, connected by *isa* relationships. A single entity consists of *components* from one or more of these entity sets, and each component is inherited from its parent.

2.5 Weak Entity Sets

It is possible for an entity set's key to be composed of attributes, some or all of which belong to another entity set. There are two reasons why we need weak entity sets.

1. Sometimes, entity sets fall into a hierarchy based on classifications unrelated to the *isa* hierarchy. If entities of set R are subunits of entities in set F, it is possible that the names of R-entities are not unique until we take into account the name of its S-entity.[2]
2. The second reason is that we want to eliminate multiway relationships, which are not common in practice anyways. These weak entity sets have no attributes and have keys purely from its supporting sets.

Definition 2.6 (Weak Entity Set)

A **weak entity set** R (double rectangles) depends on other sets. It is an entity set that

1. has a key consisting of 0 or more of its own attributes, and
2. has key attributes from **supporting entity sets** that are reached by many-one **supporting relationships** (double diamonds) from it to other sets S.

It must satisfy the following.

1. The relationship T must be binary and many-one from R to S.
2. T must have referential integrity from R to S (since these are keys and therefore must exist in supporting sets), which is why we have a rounded arrow.
3. The attributes that S supplies for the key of R must be key attributes of S, unless S is also weak, and it will get keys from its supporting entity set.
4. If there are several different supporting relationships from R to the same S, then each relationship is used to supply a copy of the key attributes of S to help form the key of R.

If an entity set supplies any attributes for its own key, then those attributes will be underlined.

[2] Think of university rooms in different buildings.

Example 2.14

To specify a location, it is not enough to specify just the seat number. The room number, and the building name must be also specified to provide the exact location. There are no extra attributes needed for this subclass, which is why a *isa* relationship doesn't fit into this.

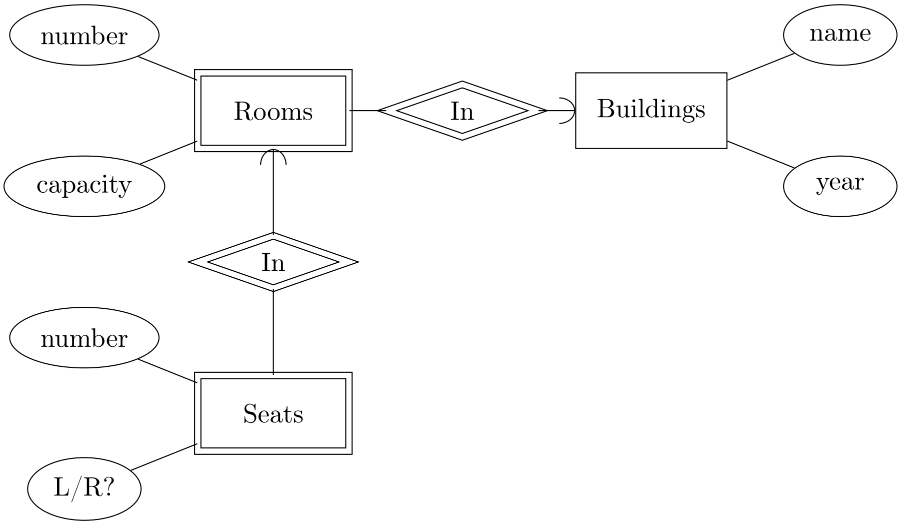


Figure 13: Specifying a seat is not enough to determine the exact location in a university. We must know the room number and the building to fully identify it. Note that we must keep linking until we get to a regular, non-weak entity.

We generally want to use a weak entity set if an entity does not have attributes to define itself.

Example 2.15

Say that we want to make a database with the constraints.

1. For states, record the name and capital city.
2. For counties, record the name, area, and location (state)
3. For cities, record the name, population, and location (county and state)
4. Names of states should be unique.
5. Names of counties are unique within a state.
6. Names of cities are unique within a county.
7. A city is always located in a single county.
8. A county is always located in a single state.

Then, our ER diagram may look like

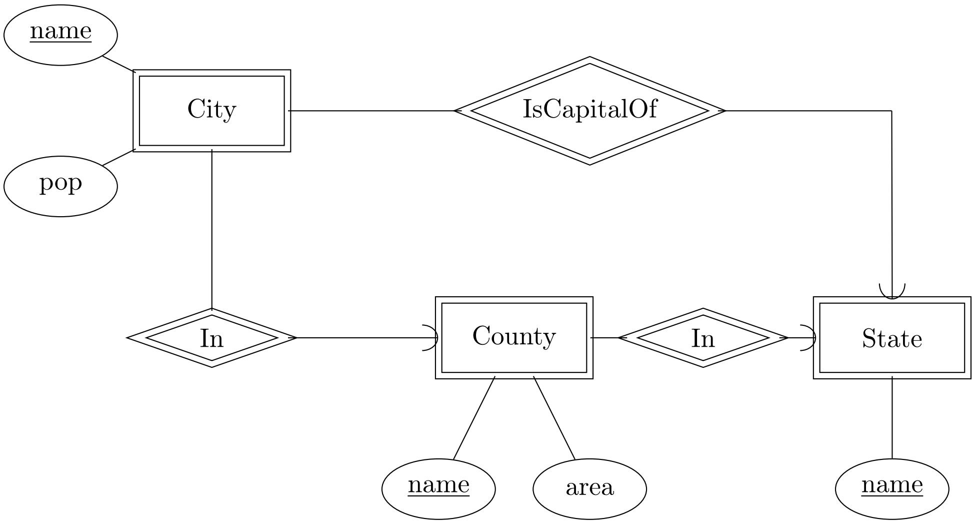


Figure 14: A weakness is that this doesn't prevent a city in state X from being the capital of another state Y.

Example 2.16

Design a database with the following.

1. A station has a unique name and address, and is either an express station or a local station.
2. A train has a unique number and engineer, and is either an express or local train.
3. A local train can stop at any station.
4. An express train only stops at express stations.
5. A train can stop at a station for any number of times during a train.
6. Train schedules are the same every day.

Then, our ER diagram may look like

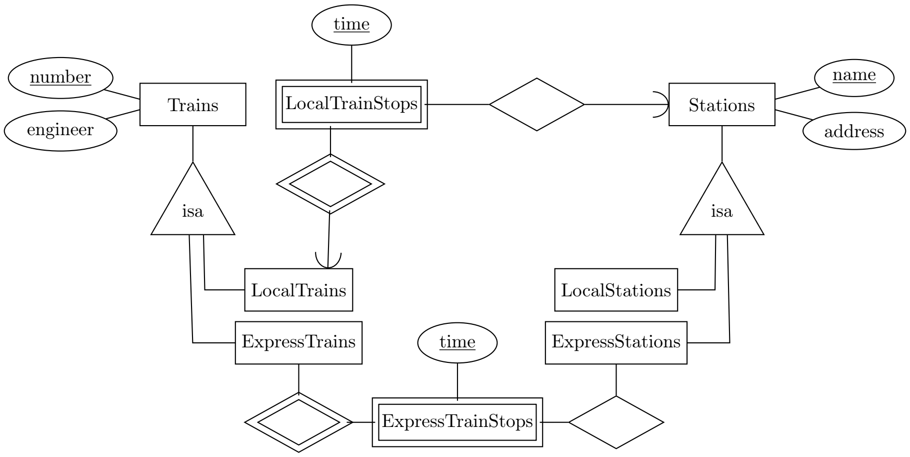


Figure 15

2.6 Translating ER Diagrams to Relational Designs

One a simple level, converting an ER diagram to a relational database schema is straightforward. Here are some rules we list.

Theorem 2.8 (Converting Entity Sets)

Turn each entity set into a relation with the same set of attributes.

Theorem 2.9 (Converting Relationships)

Replace a relationship by a relation whose attributes are the keys for the connected entity sets along with its own attributes. If an entity set is involved several times in a relationship, then its key attributes are repeated, so you must rename them to avoid duplication.

Theorem 2.10 (Reduce Repetition for Many-One Relationships)

We can actually reduce repetition for many-one relationships. For example, if there is a many-one relationship T from relation R to relation S, then $\mathbf{r}$ functionally determines $\mathbf{s}$, so we can combine them into one relation consisting of

1. all attributes of R.
2. key attributes of S.
3. Any attributes belonging to relationship T.

Theorem 2.11 (Handling Weak Entity Sets)

To build weak entity sets, we must do three things.

1. The relation for weak entity set W must include its own attributes, all key (but not non-key) attributes of supporting entity sets, and all attributes for supporting relationships for W.
2. The relation for any relationship where W appears must use the entire set of keys gotten from W and its supporting entity sets.
3. Supporting relationships should not be converted since they are many-one, so we can use the reduce repetition for many-one relationships rule above.

Example 2.17

To translate the seat, rooms, and buildings diagram,

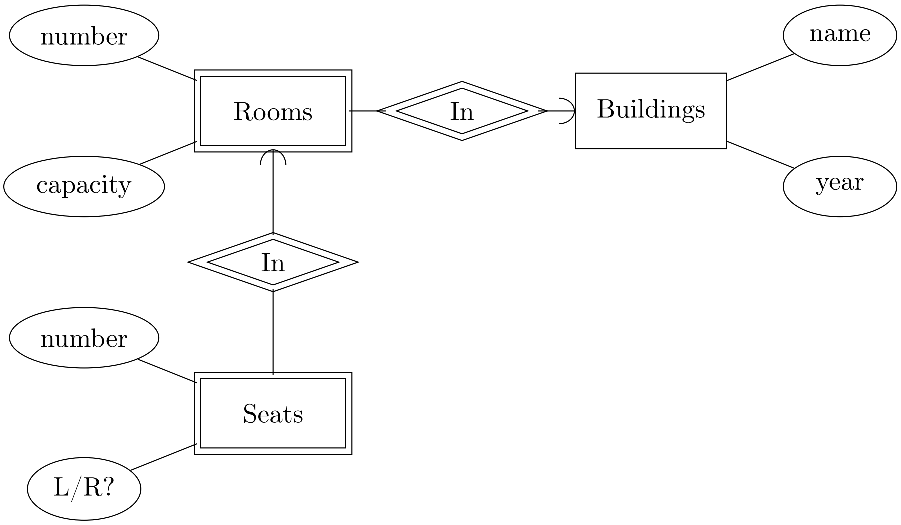


Figure 16

we have

1. `Building(`<u>`name`</u>`, year)`
2. `Room(`<u>`building_name`</u>`, `<u>`room_num`</u>`, capacity)`
3. `Seat(`<u>`building_name`</u>`, `<u>`room_num`</u>`, `<u>`seat_num`</u>`, left_or_right)`

Note that we do not need to convert the relationships since they are contained within the entity set relations. So ignore double diamonds.

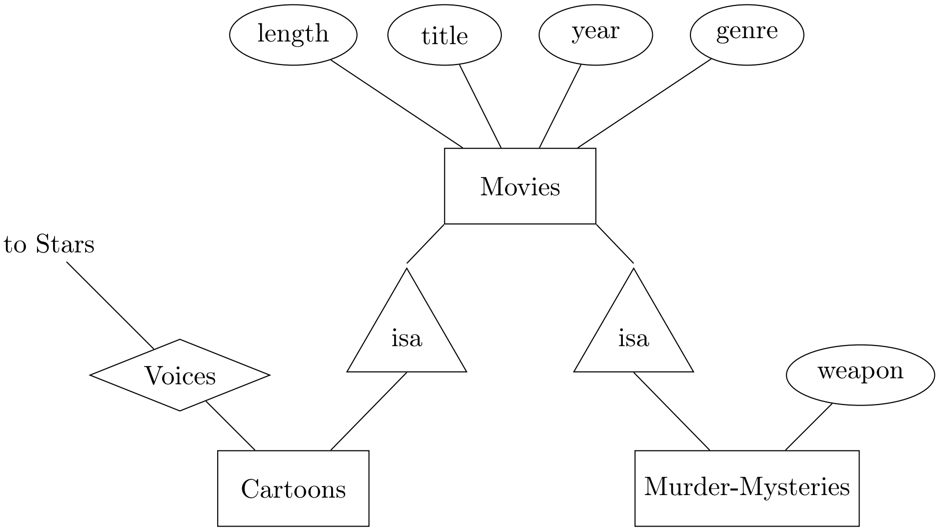


Figure 17: A figure of the movie hierarchy for convenience.

Theorem 2.12 (Converting Subclass Structures)

To convert subclass structure with a *isa* hierarchy, there are multiple ways we can convert them.

1. *E/R Standard.* An entity is in all superclasses and only contains the attributes its own subclass. For each entity set R in the hierarchy, create a relation that includes the key attributes from the root and any attributes belonging to R. This gives us

   (a) `Movies(title, year, length, genre)`
   (b) `MurderMysteries(title, year, weapon)`
   (c) `Cartoons(title, year)`
2. *Object Oriented.* For each possible subtree that includes the root, create one relation whose schema includes all the attributes of all entity sets in the subtree.
   (a) `Movies(title, year, length, genre)`
   (b) `MoviesC(title, year, length, genre)`
   (c) `MoviesMM(title, year, length, genre, weapon)`
   (d) `MoviesCMM(title, year, length, genre, weapon)`
   Additionally, the relationship would be `Voices(title, year, starName)`.
3. *Null Values.* Create one relation for the entire hierarchy containing all attributes of all entity sets. Each entity is one tuple, and the tuple has null values for attributes the entity does not have. We would in here always have a single schema.
   (a) `Movie(title, year, length, genre, weapon)`

Note that the difference between the first two is that in ER, `MurderMysteries` does not contain the attributes of its superclass, while in OO, it does.

As you probably notice, each standard has pros and cons. The nulls approach uses only one relation, which is simple and nice. To filter out over all movies, E/R is nice since we only filter through `Movies`, whilst in OO we have to go through all relations. However, when we want to filter movies that are both Cartoons and Murder Mysteries, then OO is better since we can only select from `MoviesCMM` rather than having to go through multiple relations for ER or filter out with further selections in Null. Also, OO uses the least memory, since it doesn't waste space on null values on attributes.

Example 2.18

Let's put this all together to revisit the train station example.

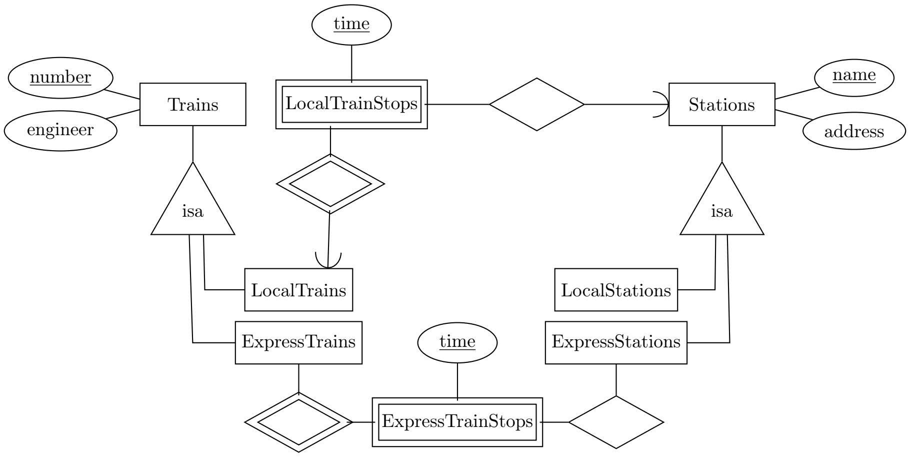


Figure 18: For convenience

We can use the ER standard to define the first 6 regular relations in single rectangles.

1. `Train(number, engineer)`
2. `LocalTrain(number`
3. `ExpressTrain(number`
4. `Station(name, address)`
5. `LocalStation(name)`
6. `ExpressStation(name)`

Then we can construct the weak entity sets.

1. `LocalTrainStops(local_train_num, time)`
2. `ExpressTrainStops(express_train_num, time)`

Then we can construct the relationships (marked with the arrows).

1. `LocalTrainStopsAtStation(local_train_number, time, station_name)`
2. `ExpressTrainStopsAtStation(express_train_number, time, express_station_name)`

Note that we can simplify these 10 relations to 8. For example, the `LocalTrain` and `LocalStation` relations are redundant since it can be computed as

$$LocalTrain = \pi_{number}(Train) - ExpressTrain \tag{36}$$

$$LocalStation = \pi_{number}(Station) - ExpressStation \tag{37}$$

There is a tradeoff since it's an extra computation when checking. However, if we had used the Null Value strategy, this would be a lot simpler, and we can use value constraints on the train and station type, which can be implemented in the DBMS (though not directly in the ER diagram).

3 SQL

SQL (Structured Query Language) is the standard query language supported by most DBMS and is a syntactically sugared form of relational algebra. It is **declarative**, where the programmer specifies what answers a query should return,but not how the query should be executed. The DBMS picks the best execution strategy based on availability of indices, data/workload characteristics, etc. (i.e. provides physical data independence). It contrasts to a **procedural** or an **operational** language like C++ or Python. The reason we need this specific query language dependent on relational algebra is that it is *less* powerful than general purpose languages like C or Python. These things can all be stored in structs or arrays, but the simplicity allows the compiler to make huge efficiency improvements.

Just like how we approached relational algebra with structure, operations, and constraints, we will do the same with SQL, although we will introduce structure with constraints first (it's natural to talk about constraints with structure, since you define the constraints when you create the SQL relations). While this wasn't talked about much before, SQL actually uses *multisets*, or *bags*, by default. It's operations support both set and bag operations, and we will cover both of them here. Working with sets or bags is really context dependent, but it does have a few advantages.

1. To take the union of two bags, we can just add everything into the other without going through to check for duplicates.
2. When we project relations as bags, we also don't need to search through all pairs to find duplicates.

This allows for efficiency at the cost of memory.

3.1 Structure and Constraints

Definition 3.1 (Primitive Types)

The primitive types are listed.[a]

1. *Characters*. `CHAR(n)` represents a string of fixed length n, where shorter strings are padded, and `VARCHAR(n)` is a string of variable length up to n, where an endmarker or string-length is used.
2. *Bit Strings*. `BIT(n)` represents bit strings of length n. `BIT VARYING(n)` represents variable length bit strings up to length n.
3. *Booleans*. `BOOLEAN` represents a boolean, which can be `TRUE`, `FALSE`, or `UNKNOWN`.
4. *Integers*. `INT` or `INTEGER` represents an integer.
5. *Floating points*. `FLOAT` or `REAL` represents a floating point number, with a higher precision obtained by `DOUBLE PRECISION`.
6. *Datetimes*. `DATE` types are of form `'YYYY-MM-DD'`, and `TIME` types are of form `'HH:MM:SS.AAAA'` on a 24-hour clock.
7. *Null*. `NULL` represents a null value.

[a] One thing to note is that keywords are usually written in uppercase by convention.

Before we can even query or modify relations, we should know how to make or delete one.

Theorem 3.1 (`CREATE TABLE`, `DROP TABLE`)

We can create and delete a relation using `CREATE TABLE` and `DROP TABLE` keywords and inputting the schema.

```
CREATE TABLE Movies(
  name CHAR(30),
  year INT,
  director VARCHAR(50),
  seen DATE
);

DROP TABLE Movies;
```

Theorem 3.2 (DEFAULT)

We can also determine default values of each attribute with the DEFAULT KEYWORD.

```
ALTER TABLE Movies ADD rating INT 0;
...
CREATE TABLE Movies(
  name CHAR(30) DEFAULT 'UNKNOWN',
  year INT DEFAULT 0,
  director VARCHAR(50),
  seen DATE DEFAULT '0000-00-00'
);
```

3.1.1 Nulls

We are not guaranteed that we will have all data. What if there are some null values? We need some way to handle unknown or missing attribute values. One way is to use a default value (like -1 for age), but this can mess with other operations, such as getting average values of certain groups of users, or can make computations harder since we have to first filter out users with `age=-1` before querying. Another way is to use a valid bit for every attribute. For example, `User(uid, name, age)` could map to `User(uid, name, name_valid, age, age_valid)`, but this is not efficient as well.

A better solution is to decompose the table into multiple relations such that a missing value indicates a missing row in one of the subrelations. For example, we can decompose `User(uid, name, age, pop)` to

1. `UserID(uid)`
2. `UserName(uid, name)`
3. `UserAge(uid, age)`
4. `UserPop(uid, pop)`

This is conceptually the cleanest solution but also complicates things. Firstly, the natural join wouldn't work, since compared to a single table with null values, the natural join of these tables would exclude all tuples that have at least one null value in them.

Definition 3.2 (NULL)

SQL's solution is to have a special value `NULL` indicating an unknown but not empty value. It has the following properties.

1. It holds for every domain (null is the same for booleans, strings, ints, etc.).
2. Any operations like $+, -, \times, >$... leads to a `NULL` value.
3. All aggregate functions except `COUNT` return a `NULL`. `COUNT` also counts null values.

Theorem 3.3 (Three-Valued Logic)

Here is another way to implement the unknown logic with *three-valued logic*. Suppose we set True=1, False=0. Then we can see that given statements x, y which evaluate to $0, 1$,

1. x and y is equivalent to $\min(x, y)$
2. x or y is equivalent to $\max(x, y)$
3. not x is equivalent to $1 - x$

It turns out that if we set unknown=0.5, then this logic also works out very nicely. Check it yourself.

The way operations treat null values is extremely important, and we will add a sentence on how it treats nulls for each operation.

Example 3.1 (Warnings)

Note that null breaks a lot of equivalences, leading to unintended consequences.

1. The two are not equivalent since if we have nulls, the average ignores all nulls, while the second query will sum up all non-nulls and divide by the count including the nulls.

```
SELECT AVG(pop) FROM User;
SELECT SUM(pop) / COUNT(*) FROM User;
```

2. The two are also not equivalent since `pop = pop` is not True, but Unknown, for nulls, so it would not return nulls. The first query would return all tuples, even nulls.

```
SELECT * from User;
SELECT * from User WHERE pop = pop;
```

3. Never compare equality with null, since this never outputs True. Rather, you should use the special keywords `IS NULL` and `IS NOT NULL`.

```
SELECT * FROM User WHERE pop = NULL; // never returns anything
SELECT * FROM User WHERE pop IS NULL; // correct
```

3.1.2 Modifying Tables

This was not an operation we defined in relational algebra, but it is so essential that we should state it now. What if we want to add or delete another attribute? This is quite a major change.

Theorem 3.4 (`ALTER TABLE`)

We can add or drop attributes by using the `ALTER TABLE` keyword followed by

1. `ADD` and then the attribute name and then its type.
2. `DROP` and then the attribute name.

```
ALTER TABLE Movies ADD rating INT;
ALTER TABLE Movies DROP director;
```

We have briefly saw how to create and drop tables. To update a table, we can do the following.

Definition 3.3 (INSERT)

You can either

1. insert one row

```
INSERT INTO Member VALUES (789, "Muchang")
```

2. or you can insert the output of a query.

```
INSERT INTO Member
(SELECT uid, name FROM User);
```

Definition 3.4 (DELETE)

You can either

1. delete everything from a table (but not the schema, unlike `DROP TABLE`).

```
DELETE FROM Member;
```

2. Delete according to a `WHERE` condition

```
DELETE FROM Member
WHERE age < 18;
```

3. Delete according to a `WHERE` condition extracted from another query.

```
DELETE FROM Member
WHERE uid IN (SELECT uid FROM User WHERE age < 18);
```

Definition 3.5 (UPDATE)

You can either

1. Update a value of an attribute for all tuples.

```
UPDATE User
SET pop = (SELECT AVG(pop) from User);
```

2. Update a value of an attribute for all tuples satisfying a `WHERE` condition.[a]

```
UPDATE User
SET name = ’Barney’
WHERE uid = 182;
```

[a] Note that this does not incrementally update the values. It updates all at once from the average of the old table from the subquery.

3.1.3 Constraints

We mainly use constraints to protect the data integrity and relieve the coder from responsibility. It also tells the DBMS about the data so it can optimize better.

Definition 3.6 (NOT NULL Constraints)

This tells that an attribute cannot be entered as null (this is already enforced for keys).

```
CREATE TABLE User
(uid INTEGER NOT NULL,
name VARCHAR(30) NOT NULL,
twitterid VARCHAR(15) NOT NULL,
age INTEGER,
pop FLOAT);
```

Definition 3.7 (PRIMARY KEY, UNIQUE Key Constraints)

There are multiple ways to identify keys.

1. Use the `PRIMARY KEY` keyword to make `name` the key. It can be substituted with `UNIQUE`. You can include at most one primary key, but any number of `UNIQUE`. This means that either name or id can be used as a key, but we must choose one primary key, so we are restricted to at most one.

```
CREATE TABLE Movies(
  name CHAR(30) NOT NULL PRIMARY KEY,
  id CHAR(30) NOT NULL UNIQUE,
  year INT NOT NULL,
  director VARCHAR(50),
  seen DATE
);
```

2. Use the `PRIMARY KEY` keyword, which allows you to choose a combination of attributes as the key. It can be substituted with `UNIQUE`.

```
CREATE TABLE Movies(
  name CHAR(30),
  year INT,
  director VARCHAR(50),
  seen DATE,
  PRIMARY KEY (name, year)
);
```

Definition 3.8 (Referential Integrity)

Like we said before, a referential integrity means that if an attribute a appears in R, then a must appear in some other T. That is, there are no **dangling pointers**, where $R.a$ does exists but $T.a$ does not. There are names for these:

1. **Foreign keys** is like $R.a$, where it must point to a valid primary key, e.g. `User.uid` or `Group.gid`, which are like entity sets.
2. **Primary keys** are like $T.a$, where it must exist if $R.a$ exists, e.g. `Member.uid` or `Member.gid`, which are relationships.

In SQl, we must make sure that the referenced columns must be the primary key and the referencing columns form a foreign key. There are two ways to do it.

```
CREATE TABLE Member (
uid INT NOT NULL
```

```
  REFERNECES User(uid),  // 1. put the references as you define it
gid CHAR(10) NOT NULL,   // 2. define the attribute first
PRIMARY KEY(uid, gid),
FOREIGN KEY (gid) REFERENCES Group(gid)); // 2. then reference it
```

If you have multi-attribute referential integrity, the second method is better.

```
...
FOREIGN KEY (gid, time) REFERENCES Group(gid, time));
```

Example 3.2 (Handling Referential Integrity Violations)

Say that you have a referential integrity constraint as above. Then there are two scenarios. If we insert or update a Member row so it refers to a non-existent User.uid, the DBMS will not allow this. If we delete or update a User row whose uid is referenced by some Member row, then there are certain scenarios.

1. *Reject.* The DBMS will reject this.
2. *Cascade.* It will ripple changes (on an update) to all referring rows.
3. *Set null.* It will set all references to NULL.

These options can be specified in SQL.

Definition 3.9 (Tuple/Attribute-Based Checks)

These checks are only associated with a single table and are only checked when a tuple/attribute is inserted/updated. It rejects if the condition evaluates to False, but *True/Unknown are fine* (unlike only True in WHERE conditions!). There are two ways to write this in SQL.

```
CREATE TABLE User(
  ... // 1. Directly put the check constraint in definition
  age INTEGER CHECK(age IS NULL OR age > 0),
  ...
);

CREATE TABLE Member(
  uid INTEGER NOT NULL,           // 2. First define attribute
  CHECK(uid IN (SELECT uid FROM User)),  // 2. Then check it
...
)
```

Note that in the second example, this is sort of like a referential integrity constraint. However, this is weaker since it only checks for changes in the Member relation, while the referential integrity constraint is checked for every change in both the Member and User relations. If a check evaluates to False, then the DBMS rejects the insertions/updates.

3.2 Basic Relational Algebra Operations

We won't repeat the definitions of the operations as they are mentioned in the first chapter. We will cover SQL syntax for both set and bag operations. Note that the way set operations work in SQL is that they remove duplicates in the operands and then remove duplicates in the result. In bag operations, we can think of each row a having an implicit count of times c_a it appears in the table.

3.2.1 Renaming

Counterintuitively, the first thing we should know how to do is rename. This is because when we do the rest of these operations, due to naming conflicts, the need to copy a relation, or reduce confusion, we often rename relations and attributes within these queries.

Definition 3.10 (Renaming)

There are two ways you can rename an attribute or a relation.

1. Explicit, using the `AS` keyword.

```
old_name AS new_name
```

2. Implicit, using a space.

```
old_name new_name
```

Example 3.3

Here's a more informative example, using syntax that we will learn later. Note that by renaming the relation, all attributes are renamed as well.

```
SELECT
    s.id,
    s.name AS student_name,
    c.name AS course_name
FROM
    Students s          -- rename relation
    JOIN Courses c      -- rename relation
    ON s.course_id = c.id;
```

3.2.2 Set/Bag Operations

Definition 3.11 (Union)

The **UNION** and `UNION ALL` keywords calculate the set and bag union, respectively. Bag union sums up the counts from the 2 tables.

```
(SELECT * FROM Relation1)  // {1, 2, 3}
UNION
(SELECT * FROM Relation2); // {2, 3, 4}
// {1, 2, 3, 4}
```

```
(SELECT * FROM Relation1)  // {1, 1, 2}
UNION ALL
(SELECT * FROM Relation2); // {1, 2, 2}
// {1, 1, 1, 2, 2, 2}
```

`UNION` and `UNION ALL` both treat NULLs as equal to each other.

Definition 3.12 (Intersection)

The **INTERSECT** and `INTERSECT ALL` keywords calculate the set and bag intersection, respectively. Bag intersection does a proper-subtract[a]

```sql
(SELECT * FROM Relation1)  // {1, 2, 3}
INTERSECT
(SELECT * FROM Relation2); // {2, 3, 4}
```

[a] Subtracts the counts and truncates counts to 0 if negative. So $\{a, a\} - \{a, a, a\} = \{\}$.

```sql
(SELECT * FROM Relation1)  // {1, 1, 2}
INTERSECT ALL
(SELECT * FROM Relation2); // {1, 2, 2}
// {1, 2}
```

Both treat nulls as equal to each other and will return NULL if it appears on both sides.

Definition 3.13 (Difference)

The **EXCEPT** and `EXCEPT ALL` keywords calculate the set and bag difference, respectively. Bag difference takes the minimum of the two counts.

```sql
(SELECT * FROM Relation1)  // {1, 2, 3}
EXCEPT
(SELECT * FROM Relation2); // {2, 3, 4}
```

```sql
(SELECT * FROM Relation1)  // {1, 1, 2}
EXCEPT ALL
(SELECT * FROM Relation2); // {1, 2, 2}
// {1}
```

Both treat NULLs as equal to each other and will return NULL if it appears more times in the first table than the second.

Example 3.4 (Interpretation of Except All)

Look at these two operations on the schema `Poke(uid1, uid2, timestamp)`.

```sql
(SELECT uid1 FROM Poke)
EXCEPT
(SELECT uid2 FROM Poke);
```

```sql
(SELECT uid1 FROM Poke)
EXCEPT ALL
(SELECT uid2 FROM Poke);
```

The first operation returns all users who poked others but were never poked, while the second returns all users who poked others more than they were poked.

3.2.3 Predicates, Projection, Selection

Definition 3.14 (Condition/Predicate)

The condition p used in in selection and joins is implemented with the `WHERE` keyword. The operations used in predicates are listed below.

1. Equality is denoted with `=`, and not equals is denoted with `<>`. Equality evaluates to true if both attribute values are not NULL and match.
2. Comparisons is done with `>`, `<`, `>=`, `<=`.
3. The `IN` and `NOT IN` gives you filters that restrict the domain of a certain attribute.

```sql
SELECT *
FROM relation
WHERE sex in ['male', 'female'];
```

`WHERE` will only select rows for which the condition is True (not False or unknown).

Both selection and projection, known as **queries**, is done with the same keyword SELECT, and we can do both of them at the same time (the DBMS will determine which operation to do first). Unlike set/bag operations before, SELECT is by default a bag operator.

Definition 3.15 (Selection)

The SELECT and SELECT DISTINCT keywords apply the selection bag and set operator, respectively. The ∗ species all attributes in the relation.

```
SELECT *
FROM relation
WHERE
  p_1 AND p_2 AND ... ;
```

Definition 3.16 (Projection)

The SELECT and SELECT DISTINCT keywords apply the projection bag and set operator, respectively. Rather than putting an ∗, we should now specify our desired attributes.

```
SELECT date, age, uid
FROM relation
```

Example 3.5 (Comparing Null values always returns Null)

The two are also not equivalent since pop = pop is not True, but Unknown, for nulls, so it would not return nulls. The first query would return all tuples, even nulls.

```
SELECT * from User;
SELECT * from User WHERE pop = pop;
```

Example 3.6 (Never Compare Equality with Null)

Never compare equality with null, since this never outputs True. Rather, you should use the special keywords IS NULL and IS NOT NULL.

```
SELECT * FROM User WHERE pop = NULL; // never returns anything
SELECT * FROM User WHERE pop IS NULL; // correct
```

3.2.4 Products and Joins

All products and joins use bag semantics and to use set semantics just use a SELECT DISTINCT.

Definition 3.17 (Cartesian Product)

The CROSS JOIN keyword or the , operator is used to calculate the cartesian product of two relations using bag semantics.

```
SELECT *
```

```sql
FROM table1
CROSS JOIN table2;

SELECT *
FROM table1, table2;
```

Definition 3.18 (Theta Join)

A theta join is performed by simply adding a `WHERE` clause after a cross join, using bag semantics.

```sql
SELECT *
FROM table1, table2
WHERE table1.A = table2.B;
```

Definition 3.19 (Natural/Inner Join)

The `JOIN` keyword returns records that have matching values in both tables using bag semantics.

```sql
SELECT * FROM Students s
JOIN Courses c ON s.id = c.id;
```

For inner join, note that NULL attributes do not match each other (as stated in equality predicate) so they are not included.

Definition 3.20 (Left Outer Join)

`LEFT (OUTER) JOIN`: Returns all records from the left table R, and the matched records from the right table S. For a given $r \in R$, if there are no $s \in S$ that matches it, then the S attributes will all be NULL.

```sql
SELECT * FROM Students s
LEFT JOIN Courses c ON s.id = c.id;
```

This keeps all NULLs on the left table.

Definition 3.21 (Right Outer Join)

`RIGHT (OUTER) JOIN`: Returns all records from the right table S, and the matched records from the left table R with potential NULLs. For a given $s \in S$, if there are no $r \in R$ that matches it, then the R attributes will all be NULL.

```sql
SELECT * FROM Students s
RIGHT JOIN Courses c ON s.id = c.id;
```

This keeps all NULLs on the right table.

Definition 3.22 (Full Outer Join)

`FULL (OUTER) JOIN`: Returns all records when there is a match in either left or right table with potential NULLs.

```
SELECT * FROM Students s
FULL OUTER JOIN Courses c ON s.id = c.id;
```

This keeps all NULLs on the both tables.

Example 3.7 (Motivation for Outer Join)

Take a look at the following motivating example. Suppose we want to find all members and their respective groups from `Group(gid, name)`, `Member(uid, gid)`, `User(uid, name)`. Then we can write the query

```
SELECT g.gid, g.name AS gname,
    u.uid, u.name AS uname
FROM Group g, Member m, User u
WHERE g.gid = m.gid AND m.uid = u.uid;
```

This looks fine, but what happens if *Group* is empty? That is, there is a group in the *Group* relation but does not appear in the *Member* relation. Then, `m.gid` will evaluate to False and would not appear in the joined table, which is fine, but what if we wanted to make sure all groups appeared in this master membership table? If a group is empty, we may want it to just have null values for `uid` and `uname`. In this case, we want to use outer join.

Example 3.8 (Complex Example with Beers)

Given the schemas

1. `Frequents(drinker, bar, times_a_week)`,
2. `Serves(bar, beer, price)`,
3. `Likes(drinker, beer)`,

say that we want to select drinkers and bars that visit the bars at least 2 times a week and the bars serves at least 2 beers liked by the drinker and count the number of beers served by the bars that are liked by the drinker. The query is shown below.

```
SELECT F.drinker, F.bar, COUNT(L.beer)
FROM Frequents F, Serves S, Likes L
WHERE F.drinker = L.drinker
  AND F.bar = S.bar
  AND L.beer = S.beer
  AND F.times_a_week >= 2
GROUP BY F.drinker, F.bar
HAVING COUNT(L.beer) >= 2
```

3.3 Arithmetic and Sorting

We can also do arithmetic on columns.

Definition 3.23 (Arithmetic Operations)

SQL supports

1. *Addition* using +. It returns NULL if at least 1 of the operands are NULL.
2. *Subtraction* using -. It returns NULL if at least 1 of the operands are NULL.
3. *Multiplication* using . It returns NULL if at least 1 of the operands are NULL.
4. *Division* using /. It returns NULL if at least 1 of the operands are NULL or there is division by 0 (or may result in error depending on DBMS). For integers it will truncate.
5. *Modulo* using %. It returns NULL if at least 1 of the operands are NULL.

You can use parentheses to compose operations together.

Example 3.9 (Arithmetic)

Here is a comphrensive example.

```
SELECT
  price + tax as total_price,
  salary - deductions as net_salary,
  price * quantity as total_value,
  total / count as average,
  id % 2 as is_odd
FROM Orders;
```

Definition 3.24 (Sorting)

The `ORDER BY` keyword sorts the relation. In fact, it is more of a display operation since the tuples in the relation are unsorted by definition.

1. `ORDER BY attribute ASC` orders in ascending order. This the default.
2. `ORDER BY attribute DESC` orders in descending order.

3.4 Aggregate Functions

Definition 3.25 (Standard SQL Aggregate Functions)

Aggregate functions compute something from a collection of tuples, rather than a single row.

1. `COUNT` counts the number of rows, including NULLS, in a query. `COUNT(DISTINCT att)` counts the distinct count of an attribute in a query and ignores NULLs.
2. `SUM` counts the sum of the values of an attribute in a query. It ignores NULL values.
3. `AVG` is the average. It ignores NULL values.
4. `MIN` is the minimum of an attribute. It ignores NULL values and only returns NULL if all values are NULL.
5. `MAX` is the maximum of an attribute. It ignores NULL values and only returns NULL if all values are NULL.

Example 3.10 (Calculate Popularity within Age Group)

If we want to find the number of users under 18 and their average popularity, then we can write

```
SELECT COUNT(*), AVG(pop)
FROM User
WHERE age < 18;
```

Example 3.11 (Average is Not Sum/Count)

The two are not equivalent since if we have nulls, the average ignores all nulls, while the second query will sum up all non-nulls and divide by the count including the nulls.

```
SELECT AVG(pop) FROM User;
SELECT SUM(pop) / COUNT(*) FROM User;
```

3.4.1 Group By

By default, the aggregate function operates on the entire relation. If we want more fine grained control by looking at a subset of tuples, then we can use the `GROUP BY` clause.

Definition 3.26 (`GROUP BY`)

`GROUP BY` is used when you want to group the query by equal values of the attributes. The syntax is

```
SELECT ... FROM ... WHERE ...
GROUP BY age;
```

To parse this, first form the groups based on the same values of all attributes in the group by clause. Then, output only one row in the select clause per group. We can look at the following order

1. `FROM`. Look at where we are querying from.
2. `WHERE`. Find out if this condition is satisfied to filter the main query.
3. `GROUP BY`. Group rows according to the values of the GROUP BY columns.
4. `SELECT`. Compute the select query for each group. The number of groups should be equal to the number of rows in the final output.

Note that if a query uses aggregation/group by, every column referenced in select must be either aggregated or a group by column.

Example 3.12

If we want to find the number of users in a certain age and their average popularity, for all ages, then we can write

```
SELECT age, AVG(pop)
FROM Users
GROUP BY age;
```

You don't necessarily have to report the group by attribute in the select. The two following examples are perfectly fine, though in the right query, `age` may not functionally determine `AVG(pop)`.

```
SELECT AVG(pop)
FROM User
GROUP BY age;
```

```
SELECT age, AVG(pop)
FROM User
GROUP BY age, name;
```

In the example below, this left query is not syntactically correct since `name` is not in the group by clause or aggregated. This is true even if `age` functionally determines `name`. Neither is the right since the lack of a group by clause means that the aggregate query is over the entire relation, which has multiple uid values.

```
SELECT age, name, AVG(pop)
FROM User
GROUP BY age;
```

```
SELECT uid, MAX(pop)
FROM User;
.
```

As you can see, this is great to use for aggregate functions. If there is no group by clause, this is equivalent to grouping everything together.

Example 3.13

Given the relation

A	B	C
1	1	10
2	1	8
1	1	10
2	3	8
2	1	6
2	2	2

Table 7: Original relation.

Running the query

```
SELECT A, B, SUM(C) AS S
FROM R
GROUP BY A, B;
```

gives

A	B	S
1	1	20
2	1	14
2	3	8
2	2	2

Table 8: Our query.

3.4.2 Having

Definition 3.27 (HAVING)

If you want to filter out groups having certain conditions, you must use the `HAVING` keyword rather than `WHERE`. The syntax is

```
SELECT A, B, SUM(C) FROM ... WHERE ...
GROUP BY ...
HAVING SUM(C) < 10;
```

You should look at the `HAVING` clause after you look at the `GROUP BY` but before `SELECT`. `HAVING` will only select rows for which the condition is True (not False or null).

Example 3.14

Given the relation

A	B	C
1	1	10
2	1	8
1	1	10
2	3	8
2	1	6
2	2	2

Table 9: Original relation.

Running the query

```
SELECT A, B, SUM(C) AS S
FROM R
GROUP BY A, B
HAVING SUM(C) > 8;
```

gives

A	B	S
1	1	20
2	1	14

Table 10: Our query.

Example 3.15

Given the schema `Sailor(sid, name, age, rating)`, to find the age of the youngest sailor with age at least 18, for each rating with at least 2 sailors, we can run the query.

```
SELECT S.rating, MIN(S.age) AS minage
FROM Sailors S
WHERE S.age >= 18
GROUPBY S.rating
HAVING COUNT(*) > 1;
```

rating	age
7	45.0
1	33.0
8	55.5
8	25.5
10	35.0
7	35.0
10	16.0
9	35.0
3	25.5
3	63.5
3	25.5

(a) Original table.

rating	age
7	45.0
1	33.0
8	55.5
8	25.5
10	35.0
7	35.0
9	35.0
3	25.5
3	63.5
3	25.5

(b) Filter out ages less than 18.

rating	age
1	33.0
3	25.5
3	63.5
3	25.5
7	45.0
7	35.0
8	55.5
8	25.5
9	35.0
10	35.0

(c) Group by rating.

Figure 19: The thought process. Data tables showing the transformation of rating and age values.

3.5 Nested Queries and Views

3.5.1 Subqueries

We have so far worked with a single query consisting of a single select statement. However, we can extend this by including queries that have other queries in them. Conceptually, this is easy to understand since we are just composing more operators in relational algebra.

Definition 3.28 (Subquery)

A **subquery** is a query contained inside another query.

Definition 3.29 (Correlated Subqueries)

A **correlated subquery** is a subquery that is an input to a parent query. It is a query that needs a parameter from the main query and are generally slower to run. They are hard to parse, but generally you should look in this order (though this is not how the DBMS actually implements it).

1. `FROM`. Look at where we are querying from and for loop over each row.
2. `SELECT`. For each row in the outer query, take whatever values/parameters are needed for this row to input into the subquery.
3. `WHERE`. Find out if the condition resulting from the subquery is satisfied. If so, keep this.

Definition 3.30 (`EXISTS`)

The `EXISTS`(subquery) keyword checks if a subquery is empty or not, and `NOT EXISTS` checks the negation.

Example 3.16 (Age)

Given `User(name, age)`, say that we want to get all users whose age is equal to a person named Bart. Then, we want to select all users from the relation. For each user, we perform the subquery where this original tuple age coincides the others age and the others name is Bart. The outer query only returns those rows for which the `EXISTS` subquery returned true. Then we can write the two equivalent queries.

```
SELECT *
FROM User as u
WHERE EXISTS(SELECT * FROM User
              WHERE name = "Bart"
              AND age = u.age);
```

```
SELECT *
FROM User
WHERE age IN(SELECT age
              FROM User
              WHERE name = 'Bart');
```

To understand this, we use the rules from above.

1. FROM. We take each row in the outer query.
2. SELECT. For each row, we take the age from that row.
3. WHERE. The subquery looks for any user named Bart with that same age. If such a user exists, the row is kept.

Here is a very useful keyword that simplifies complex nested queries, one example of a **common table expression (CTEs)**.

Definition 3.31 (WITH)

The WITH clause aliases many relations returned from queries.

```
WITH Temp1 AS (SELECT ...),
     Temp2 AS (SELECT ...)
SELECT X, Y
FROM Temp1, Temp2
WHERE ...
```

Example 3.17

To extend the Bart age example, we can think of temporarily storing a query of only Bart's ages, and then comparing it when doing the main query over User.

```
WITH BartAge AS
  (SELECT age
  FROM User
  WHERE name = 'Bart')
SELECT U.uid, U.name, U.pop,
FROM User U, BartAge B
WHERE U.age = B.age;
```

3.5.2 Scalar Subqueries

Definition 3.32 (Scalar Subqueries)

Sometimes, if a query returns 1 scalar, then you can use it in your WHERE clause. You must be sure that a query will return exactly 1 scalar (not 0 and not more than 1), or there will be a runtime error.

Example 3.18

If we want to compute users with the same age as Bart, we can write

```
SELECT * FROM User
```

```
WHERE age = (SELECT age from User WHERE name = ’Bart’);
```

However, we may not know if Bart functionally determines age, so we must be careful.

3.5.3 Quantified Subqueries

Definition 3.33 (`ALL`, `ANY`)

We have the following **quantified subqueries**, which performs a broadcasting condition.

1. `ALL` checks if all conditions are true. If any are NULL, then the whole result is NULL.
2. `ANY` checks if any condition is true. If any are true, then the whole result is true. If all are either NULL or false, then `ANY` will return NULL.

Example 3.19 (Popular Users)

Which users are the most popular? We can write this in two ways.

```
SELECT *
FROM User
WHERE pop >= ALL(SELECT pop from User);
.
```

```
SELECT *
FROM User
WHERE NOT
  (pop < ANY(SELECT pop from User));
```

To review, here are more ways you can do the same query.

```
SELECT *
FROM User AS u
WHERE NOT EXISTS
  (SELECT * FROM User
  WHERE pop > u.pop);
```

```
SELECT * FROM User
WHERE uid NOT IN
  (SELECT u1.uid
  FROM User as u1, User as u2
  WHERE u1.pop < u2.pop);
```

3.5.4 Views

Definition 3.34 (View)

A **view** is a virtual table that can be used across other queries. Tables used in defining a view are called **base tables**.

Example 3.20 (Jessica’s Circle)

You can create a temporary table that can be used for future queries.

```
CREATE VIEW JessicaCircle AS
SELECT * FROM User
WHERE uid in (SELECT uid FROM Member WHERE gid = ’jes’);
```

Once you are done, you can drop this view with

```
DROP VIEW JessicaCircle;
```

3.6 Recursion

Recursion seems like a foreign concept in relational databases and was not supported until SQL3 , but we can consider the following motivating example.

Example 3.21 (Ancestors)

Given a schema `Parent(parent, child)`, we can find Bart's grandparents with the following query.

```
SELECT p1.parent as grandparent
FROM Parent p1, Parent p2
WHERE p1.child = p2.parent
AND p2.child = 'Bart';
```

We can do the same for parents, great grandparents, and so forth, but there is no clean way to get all of Bart's ancestors with a single query.

Definition 3.35 (Recursion)

You can implement a recursive query using the `WITH RECURSIVE` keyword.

Example 3.22 (Recursive Query for Ancestors)

Lines 2-7 defines the relation recursively, and then lines 8-10 is the query using the relation defined in the WITH clause.

```
1  WITH RECURSIVE
2  Ancestor(anc, desc) AS
3  ((SELECT parent, child FROM Parent) -- base case
4  UNION
5  (SELECT a1.anc, a2.desc              -- recursion step
6  FROM Ancestors a1, Ancestors a2      -- recursion step
7  WHERE a1.desc = a2.anc))             -- recursion step
8  SELECT anc
9  FROM Ancestor
10 WHERE desc = 'Bart';
```

4 Index and B+ Trees

So far, we've abstracted away the inner workings of the DBMS and was just told to trust it. From here, we'll delve into the inner workings of modern DBMS systems, starting with hardware.

4.1 Hardware and Memory Layout

Recall the memory hierarchy, which starts with CPU registers, followed by caches, memory, and a disk. The speed of read/write, called I/O, determines how fast you can retrieve this data.

Definition 4.1 (Hard Disk)

In SQL queries, the (disk) I/O dominates the execution time, so this must be analyzed first. A typical hard drive consists of a bunch of **disks/platters** that are spun by a **spindle** and read by a **disk head** on a **disk arm**. Each disk is pizza sliced into **sectors** and donut sliced into **tracks** (and a collection of tracks over all disks is a **cylinder**), and the arm will read one **block** of information, which is a logical unit of transfer consisting of one or more blocks (i.e. like a word, which is usually 4 bytes).[a]

[a] You cannot just write 1 byte. You must rewrite the entire block with the 1 byte updated. If we want to write 2 bytes which are on separate blocks, we must do 2 block writes.

Definition 4.2 (Access Time)

The access time is the sum of

1. the *seek time*: time for disk heads to move to the correct cylinder
2. the *rotational delay*: time for the desired block to rotate under the disk head
3. the *transfer time*: time to read/write data on the block

This spindle rotation and moving arm is slow, so the times are dominated by the first two.

Note that this is heavily dependent on the data being accessed. Sequential data is extremely fast while random access is slow. Therefore, we should try to store data that should be accessed together next to each other in the disk.

Definition 4.3 (Memory, Buffer Pool)

As we expect, the DBMS stores a cache of blocks, called the **memory/buffer pool** containing some fixed size of M maximum blocks. We read/write to this pool of blocks (which costs some time per blocks) and then these dirty (which are written/updated) are flushed back to the disk. It essentially acts as a middleman between the disk and the programmers, and every piece of data that goes between the two must go through the memory.

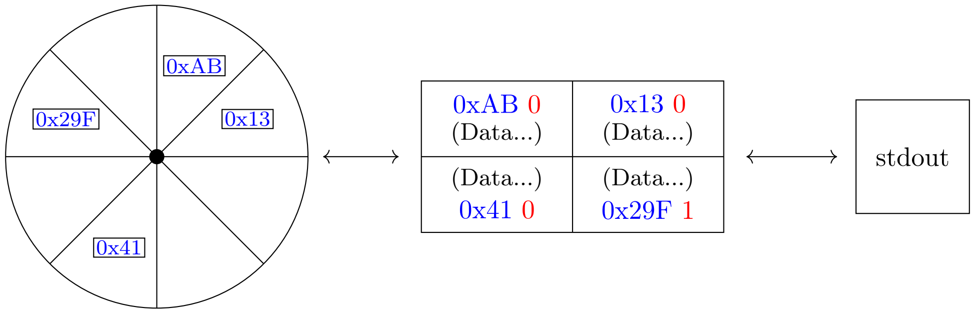


Figure 20: We store the memory address of the block (blue) and a bit indicating whether it is dirty or not (red).

Therefore, if we want to read $N > M$ blocks, then the first M blocks must be loaded into memory, outputted into stdout, and then the memory must be refilled with the rest of the blocks. If we have updated a block in memory, then we should flush it before overwriting this block in memory.[3] Replacement strategies won't be covered here. Note that unlike algorithms, which focus on the cost of the algorithm after it is read into memory, we focus on the cost of loading data from the disk into memory.

So how should we increase performance?

1. *Disk Layout*: Keep related things close together, e.g. in the same sector/block, or same track, or same cylinder, or adjacent cylinders.
2. *Prefetching*: When fetching a block from the disk, fetch the next block as well since it's pretty likely to access data from the next block. This is basically locality.
3. *Parallel I/O*: We can have more heads working at the same time.
4. *Disk Scheduling Algorithm*: e.g. the elevator algorithm sorts the cylinders so that you don't go back and forth between cylinders when fetching.
5. *Track Buffer*: Read/write one entire track at a time.

Now let's talk about how the actual bytes are stored in memory.

Definition 4.4 (Row Major Order, NSM)

We group rows together contiguously in the disk block.[a] If we have a relation with schema `R(INT(4), CHAR(24), INT(4), DOUBLE(8))`, we first store the rows together, with extra space in between since we might append new attributes, and have a tuple of pointers to the start of each row (orange lines).

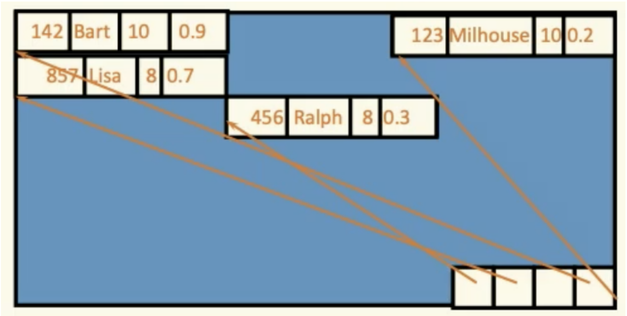


Figure 21

[a] This is the most standard storage policy.

Note that `VARCHAR` still allocates the same number of bytes, but adds padding.

Definition 4.5 (Column Major, PAX)

We group columns together contiguously in the disk block, which allows for optimization since all types are the same and we can just use pointer arithmetic to get every attribute in $O(1)$ time. We split the block into chunks and have metadata of pointers that point to the start of the array for each attribute.

[3] idk perhaps we can have a overhead bit indicating whether a block is updated, along with the memory address of this block so it can find it back on the disk.

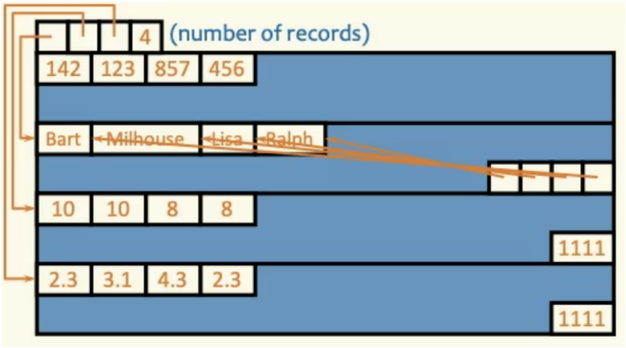


Figure 22

4.2 Search Keys and Index

Now that we've seen how the data is actually laid out in memory, we can look under the hood to see what happens when we make a query as such. Consider the relation schema `User(uid, age)`, along with the two SQL queries, specifying on the key attribute and a non-key attribute.

```
SELECT *
FROM User
WHERE uid = 112;
```

```
SELECT *
FROM User
WHERE age = 12;
```

The DBMS will have to go to disk and scan it to find the tuples satisfying the predicate. However, it is a bit more sophisticated than a simple complete scan.

Definition 4.6 (Search Key)

The **search key** is simply the attribute on which our query is defined (e.g. `uid` and `age` in the example above). The values (e.g. `112` and `12`) are called the **search key values**.

This is distinct from a closely-related concept called the index, which also refers to attributes. The tuples may take on some structure depending on the attribute. It may be sorted in one attribute but random in another.

Definition 4.7 (Index)

This set of attribute values, which really act as pointers, that we use to scan more efficiently on disk is called the **index**.

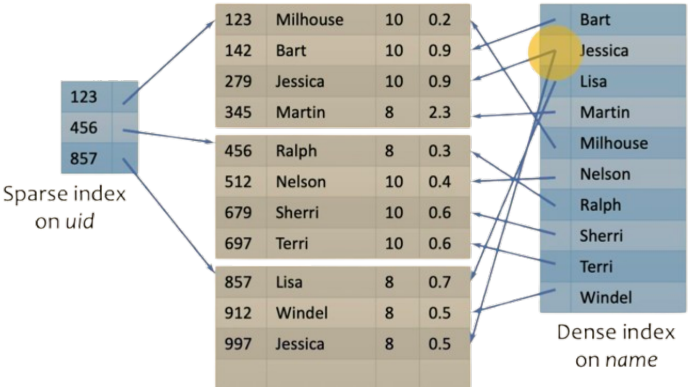


Figure 23: The dense index has 10 values, but there are two Jessicas, so it is indeed dense. The sparse index is on the uid, and since the relation is sorted (a more specific form of *clustering*) in the disk, we can use sparse keys.

Sparse and dense just refers to how much an index "covers" a disk block.

1. A **dense index** means that there is one index entry for each search key value. One entry may point to multiple records
2. A **sparse index** means that there is one index entry for each block. Records must be clustered according to the search key as shown below, and this can optimize searching.

Note that

1. The sparse index must contain at least the number of blocks, while the dense index must contain at least the number of unique search key values. Since the sparse index is much smaller, we may be able to fit it into main memory and not have to pay the I/O cost.
2. A dense index does not require anything on the records, while the sparse requires the data to be clustered.
3. Lookup is easy on dense since we can directly see if it exists. For sparse, we must first follow the pointer and scan the block. (e.g. if we wanted to look for 279, we want to look at the address pointed to by 123, and scan down until we hit it or reach a number greater than 279).
4. Update is usually easier on sparse indices since we don't need to update the index unless we add a new block. For dense, if we added a new person Muchang, then we would have to add `Muchang` to the dense index.

Definition 4.8 (Primary vs Secondary Index)

Primary and secondary refers to what the index is over.

1. A **primary index** is created for the primary key of the relation. Records are usually clustered by the primary key, so this can be sparse.
2. A **secondary index** is any index that is not over a primary key and is usually dense.

In SQL, the `PRIMARY KEY` declaration automatically creates a primary index, and the `UNIQUE` declaration creates a secondary index.

Example 4.1 (Additional Secondary Indices)

You can also create an additional secondary index on non-key attributes. For example, if you think that you will query based on popularity often, you can do

```sql
CREATE INDEX UserPopIndex ON User(pop);
```

which will create a dense index to speed up lookup at the cost of memory.

4.3 Tree Index

4.3.1 B-Trees

So really, an index is really a set of pointers, but we must be able to store this set of pointers. This leads to the problem of the index being too big to fit into a block. In this case, we can just create a sparse index on top of the dense index. This allows us to store a large index across blocks.

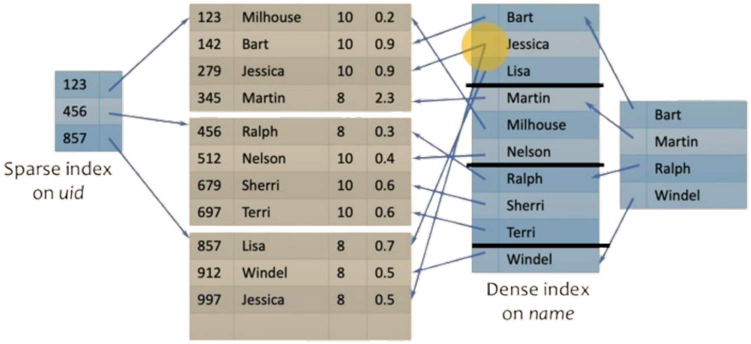


Figure 24: We store a sparse index in memory which points to a dense index on disk, which then points to the tuples of the relation, also on disk.

If the index is still too big, we store another index on top of that, and we have pretty much a tree. This is called the Index Sequential Access Method (ISAM).

Example 4.2 (Index Traversal in Tree)

If we want to look up 192 in this tree, we traverse down. Note that we have the root node in memory already, but we would require an IO operation to traverse the pointer to the first block, and then another IO cost to go to the second.

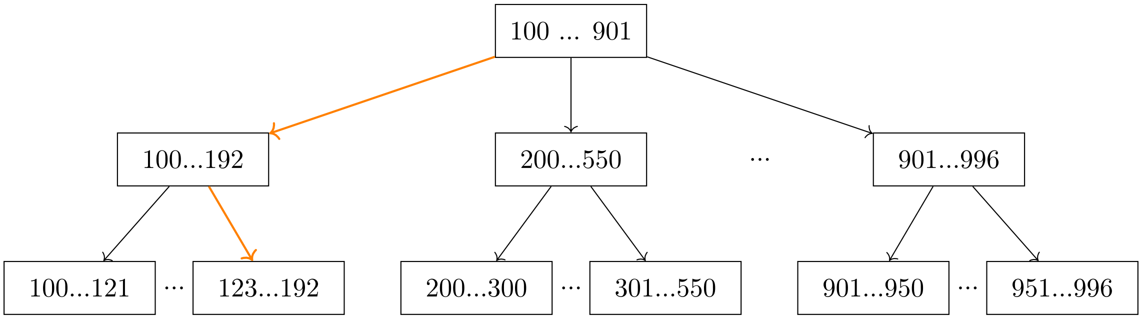


Figure 25: An index tree of depth 3. If you want to look up 192, then you would traverse down the search tree.

A problem with this method is also a problem with BSTs. If we want to delete 123 and add 121 ten times, then we have an unbalanced binary search tree and in extreme cases, this reduces to a linear search.

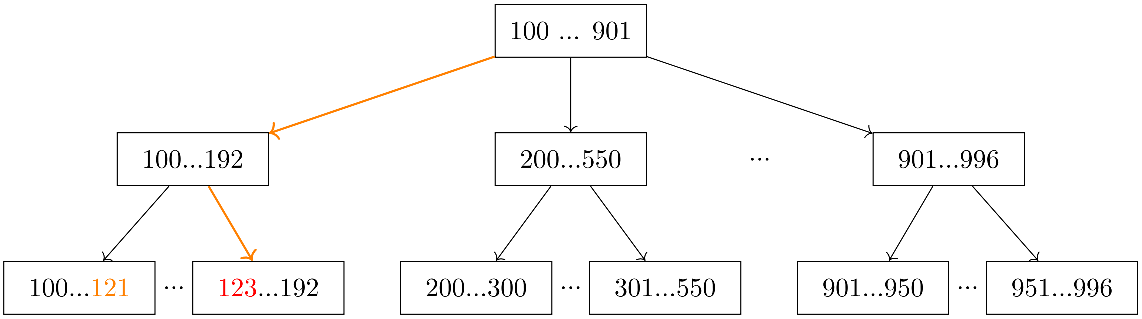


Figure 26: If this block overflows, then we want to expand this, leading to an unbalanced BST and reducing our search to linear time.

Definition 4.9 (B Tree)

Therefore, this static data structure is not good, and we must use a more flexible one, called a **B-tree**.

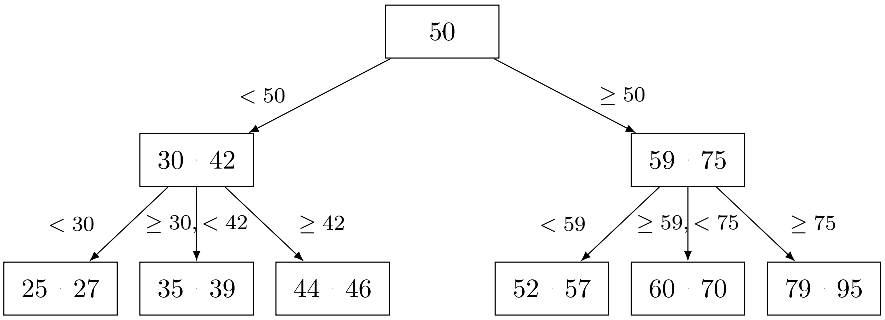


Figure 27: B+ tree structure where each node can hold multiple entries with fixed maximum size and is sorted. The tree maintains height balance, and all leaf nodes are sorted. Each node does not need to be full, and the number of pointers equals the number of entries plus one.

4.3.2 B+ Trees

The actual structure that modern DBMS uses is the slightly more sophisticated B+ tree.

Definition 4.10 (B+ Tree)

While in B trees, the non-leaf nodes are also data, **B+ trees** divide the nodes into leaf nodes, which represent the data in this data structure, and the non-leaf nodes, which do not represent data but index nodes containing index entries.

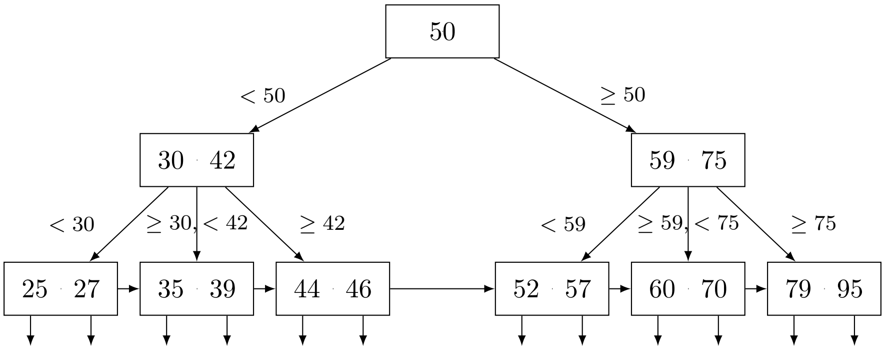
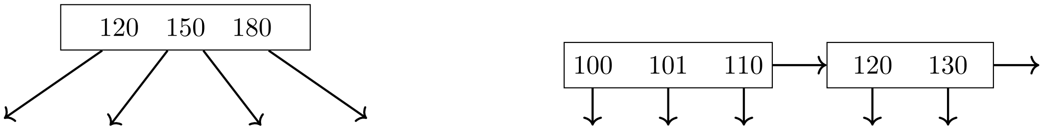


Figure 28: The main difference is that the index nodes contain index entries, and the leaves are linked. Note that the 59, which represents both the index and the data value, are repeated. In here, we assume a block size of 2. The leaf nodes are the indices that point to the tuples in the disk/memory. Sometimes, we may store the tuple directly in the nodes, which saves us another level of indirection, but this may cause memory problems if the tuples have too many attributes.

It has the following constraints.

1. The **fanout** refers the to maximum number of pointers that can come out of each node.



(a) Non-leaf node of a B-tree with four child pointers and three separator keys

(b) Leaf nodes of a B-tree with pointers to data and links to adjacent leaves

Figure 29: With fanout constraint 4, for index nodes, we have 3 values with 4 pointers representing each range. For the leaves, we have 3 pointers to the disk addresses plus 1 pointer to the next node.

2. The root is the only node that can store one value. It is special in this way.

Therefore, if we have a fanout of f, the table shows the constraints.

	Max # pointers	Max # keys	Min # active pointers	Min # keys
Non-leaf	f	$f-1$	$\lceil f/2 \rceil$	$\lceil f/2 \rceil - 1$
Root	f	$f-1$	2	1
Leaf	f	$f-1$	$\lceil f/2 \rceil$	$\lceil f/2 \rceil$

Figure 30: Note that there is a minimum constraint to ensure that the tree is balanced.

Now let's describe the implementation behind its supported operations.

Algorithm 4.0 (Lookup)

If we query `SELECT * FROM R WHERE k = 179;`, we go through the B+ tree's indices and reach the leaf node. From here, we can use the pointer to go to the memory address holding this tuple in memory/disk. If we query `k = 32`, then we will not find it after reaching `35`. If we want to query a range `32 < k < 179`, we start at 32 and use the leaf pointers to go to the next leaf until we hit 179.

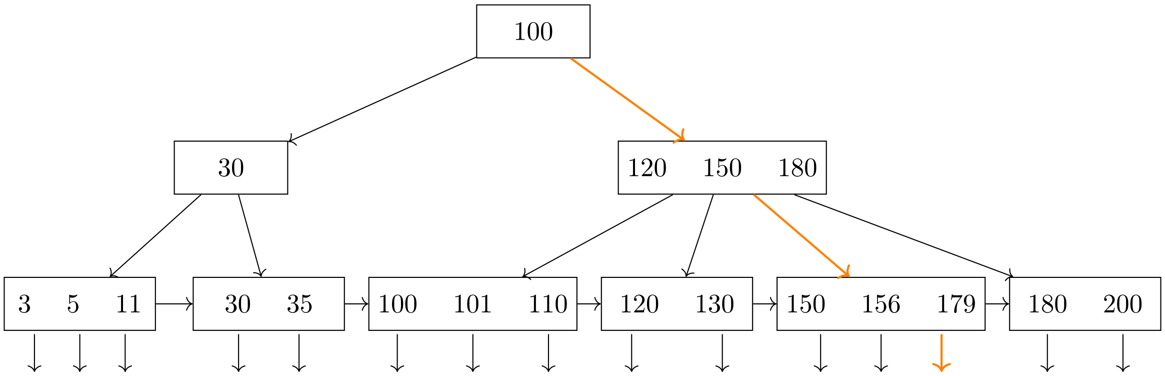


Figure 31: B-tree structure with nodes containing multiple values and pointers to child nodes. Leaf nodes have horizontal connections for sequential access and vertical pointers to data records. Note that the max fanout is 4.

In practice, there are much more pointers, so we could start at 179 and go back to 32 if we had backwards pointers.

Algorithm 4.0 (Insertion)

Insertion onto a leaf node having space is easy.

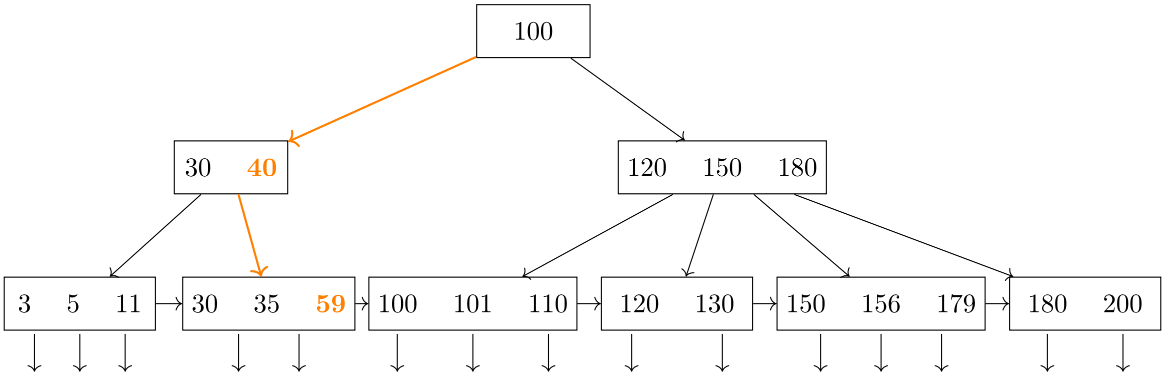


Figure 32: You can simply follow the orange to look up where the inserted ky should goNote that to traverse this tree, we had to access 3 blocks of memory since for each block, we had to go to the disk, look up its contents, and retrieve the index of the next block (retrieve the blocks containing (100), (30), and (30, 35)) Then we have to update this block of (30, 35) and flush it.

When we have a full node, it is more complicated.

1. Say that we have a full node. We can try to push the rest of the leaf nodes to next node having space, but we are not guaranteed that the neighbors will have space.

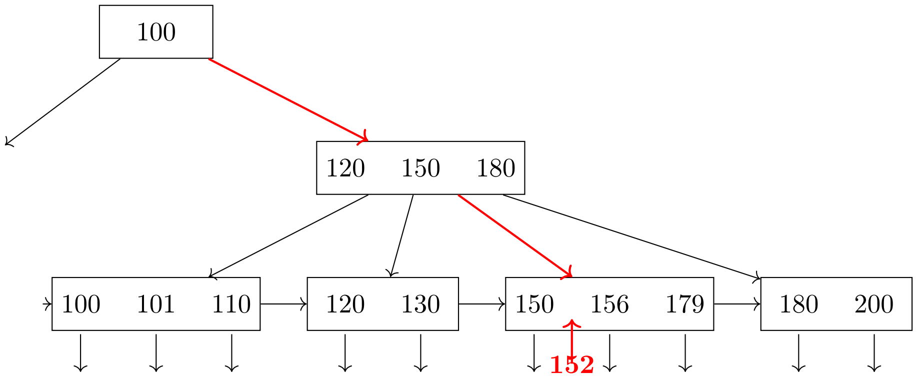


Figure 33: If you try to add 152 into the tree, then it would overflow the leaf node.

2. We can split the node, called a **copy-up**. However, this new node now has no pointer. If the parent node is not full, we can just add the pointer and update its value.

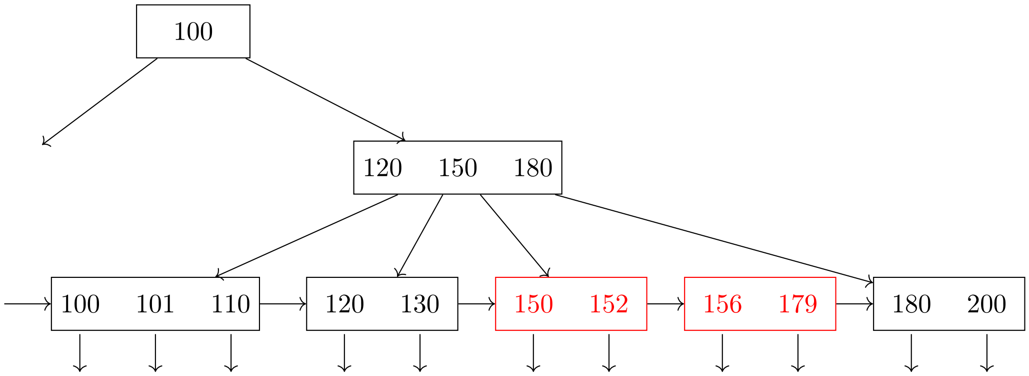


Figure 34

3. If it is full, then we should split the parent node as well. This is called a **push-up**.

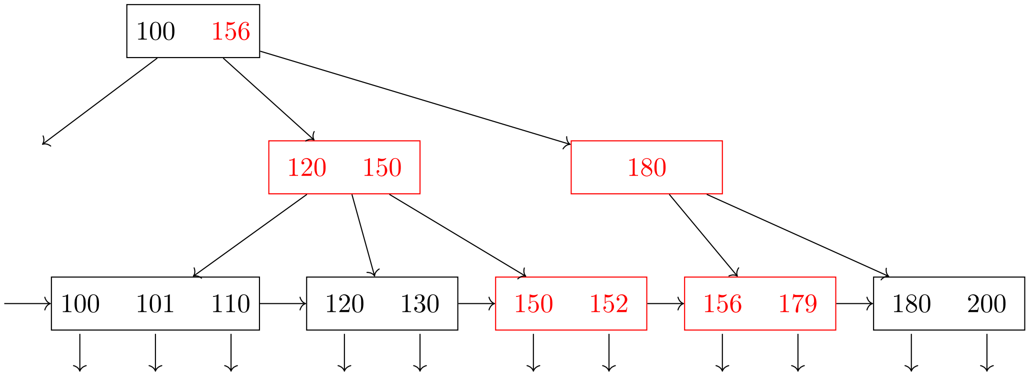


Figure 35: You need to add to the parent node a pointer to the newly created node.

4. This means that we have to update the parent of the parent. If the parent is not full, then we simply add it, and if it is full, then we split the parent of the parent, and so on. If we reach the root node, then we just split the root and increase the height of the tree.[a]

[a] This is rare in practice since the fanout is much greater than 3.

Algorithm 4.0 (Deletion)

Deleting a value is simple if after deletion, the leaf node has at least $f/2$ pointers ($f/2 - 1$ values).

1. If the node has less than $f/2$ pointers, then it is too empty. We can adjust by taking adjacent nodes and moving them from the full nodes to the empty nodes, but this may steal too much from siblings.

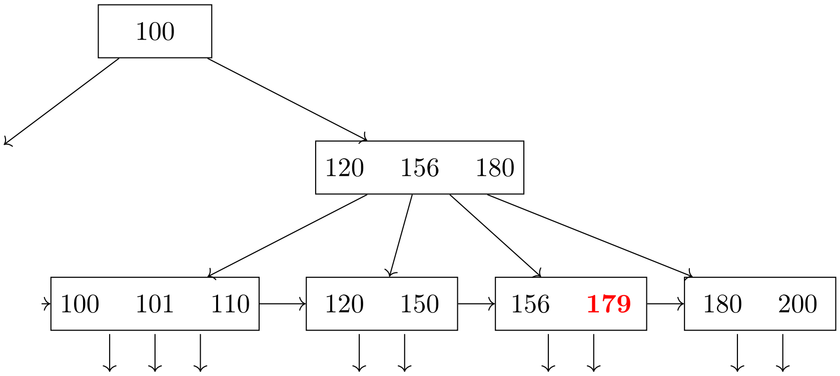


Figure 36: If you try to delete 179 and take any index from the adjacent leaf nodes, they would be too empty as well!

2. The adjacent nodes may be empty as well, and in this case we want to **coalesce** or merge the nodes together. This results in a dangling pointer, so we delete the pointer and remove a value from the parent node.[a]

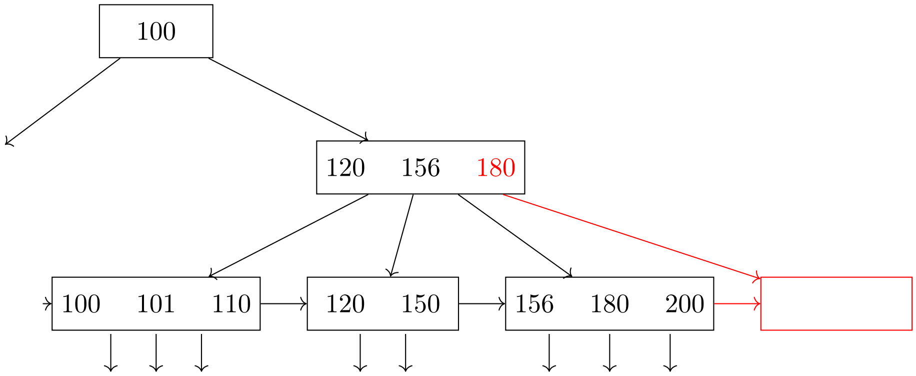


Figure 37: You should merge the right node into the node with the deleted element, and then delete the nodes and pointers that are not needed (in red).

3. We keep doing this until the B+ tree requirements are satisfied or we reach the root, at which point we delete the root node and our B+ tree height decreases by 1.

[a] In practice, this is not done every deletion. This reorganizing process can be deferred and the B+ tree property may temporarily not hold.

Theorem 4.1 (IO Cost of Lookup, Insertion, Deletion)

In general, all these operations have similar runtimes:

1. They require us to go to the bottom of the tree, so it is h operations, where h is the height.
2. We also maybe have +1 or +2 to manipulate the actual records, plus $O(h)$ for reorganization like

we saw before (which is rare if f is large).

3. Minus 1 if we cache the root in memory, which can be decreased even further if we cache more blocks.

The actual size of h is roughly $\log_{f/2} N$, where N is the number of records. f is the fanout, but the B+ tree properties guarantee that there are at least $f/2$ pointers, so it is $\log_{f/2} N$ at worst and $\log_f N$ at best. f is very large usually so this is quite good.

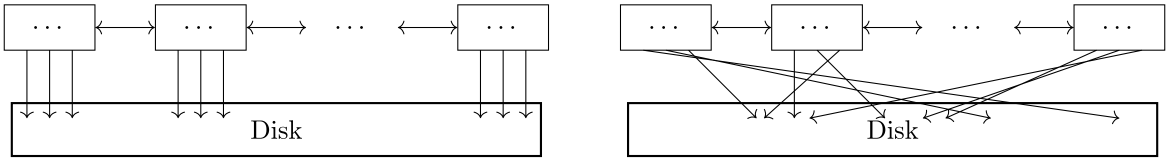
The reason we use B+ trees rather than B trees is that if we store data in non-leaf nodes, this decreases the fanout f and increases h. Therefore, records in leaves require more I/Os to access.

4.3.3 Clustered vs Unclustered Index

Note that in a B+ tree, the leaf nodes always store the index in sorted order. This is so we can have fast lookup in indices. When we go into the disk, however, we may not have this assumption.

Definition 4.11 (Clustered vs Unclustered Index)

If the order of the data records on disk is the same as or close to the order of data entries in an index, then it is **clustered**, and otherwise **unclustered**.



(a) Clustered Index

(b) Unclustered Index

Figure 38: Even if the data entries (leaf nodes) are sorted, the memory addresses of the blocks on disk that they point to may not be sorted. Note that the B+ tree is still a search tree! It is sorted. The clustered is a property of the data on disk (blue squares).

The performance can really depend on whether the index is clustered or unclustered.

4.4 Hash and Composite Index

Definition 4.12 (Hash Index)

So far, we have used tree indices. However, an alternative is to use a **hash index** which hashes the search keys for comparison. Hash indices can only handle equality queries.

1. `SELECT * FROM R WHERE age = 5;` (requires hash index on `(age)`)
2. `SELECT * FROM R, S WHERE R.A = S.A;` (requires hash index on `R.A` or `S.A`)
3. `SELECT * FROM R WHERE age = 5 AND name = 'Bob'` (requires hash index on `(age, name)`)

They are more amenable to parallel processing but *cannot handle range queries or prefixes*, e.g. `SELECT * FROM R WHERE age >= 5;`. Its performance depends on how good the hash function is (whether it distributes data uniformly and whether data has skew).

Definition 4.13 (Composite Index)

We've looked at queries in the form `SELECT  FROM User WHERE age = 50;`, but what if there are multiple conditions. For example, if we have

```
SELECT * FROM User WHERE age >= 25 AND name = 'B';
```

then we can do a couple things:

1. We have the index on `(age)`, at which point the leaf nodes will look something like
$$25, 25, 25, 26, 26, 28, 29, 29 \tag{38}$$
We traverse through all the addresses in these leaf nodes and find the ones with name `B`.
2. If we have a **composite index** on `(age, name)`, then our leaf nodes will sort them as
$$(25, A), (25, A), (25, B), (26, A), (26, C), (28, B), (29, A), (29, C) \tag{39}$$
3. If we index on `(name, age)`, then our leaf nodes will sort them as
$$(A, 25), (A, 25), (A, 26), (A, 29), (B, 25), (B, 28), (C, 26), (C, 29) \tag{40}$$
Note that if we had the query `SELECT  FROM R WHERE age >= 25`, then this sorting would not help since we do not prioritize ordering by age. So we cannot use tree indexing over name, age for this query.

Example 4.3

Therefore, given a query, there are certain indices that we can use or cannot use for it.

1. If we have a query `A >= 5`,
   (a) Can use hash index in general.
   (b) Can use tree index in general.
2. If we have query `A >= 5`,
   (a) Can use tree with index (A).
   (b) Can use tree with index (A, B).
   (c) Cannot use tree with index (B, A) since A is not prefix.
3. If we have query `A = 5`,
   (a) Can use hash with index (A).
   (b) Can use tree with index (A).
   (c) Cannot use hash with index (A, B) since hashing this tuple does not allow us to compare to A or retrieve it. It is one-way and pseudo-random.
   (d) Can use tree with index (A, B).
4. If we have `A = 5 AND B = 7`,
   (a) Can use hash with index (A, B).

Each index has its pros and cons, so why not just use both tree and hash indices? The problem is that when we modify a relation on the disk, we need to update the index as well. Therefore, having too many indices requires us to update more and takes more disk space.

Okay, so we can't use too many indices, but are indices *always* better than table scans? Not exactly.

Example 4.4 (Table Scans Wins)

Consider $\sigma_{A>v}(R)$ and a secondary, non-clustered index on $R(A)$ with around 20% of R satisfying $A > v$ (could happen even for equality predicates). We need to follow pointers to get the actual result tuples.

1. IOs for scan-based selection is simply $B(R)$ (where we can retrieve multiple tuples in this block), while
2. IOs for index-based selection is the lookup-cost (to traverse down the tree) plus $0.2|R|$ (since for each tuple, we do a IO lookup, retrieve it, and then have to retrieve the next tuple which is likely

not in the same block)
So table scan wins if a block contains more than 5 tuples since we might as well grab everything rather than look them up one by one.

Thankfully, the query optimizer will make these decisions for you.

4.5 Index Only Plans

Definition 4.14 (Index-Only Plans)

There are queries that can be answered only by accessing the index pages and not the data pages, known as **index-only plans**. For index-only plans, clustering is not important since we are looking only at the leaf nodes at most. Therefore, we only need to compute the I/O cost of traversing the tree and not to access data.
For equality, we can just compute the number of tuples in the index pages where the equality condition is satisfied. For ranges, we may need to traverse the leaf nodes, which will lead to additional I/O cost to retrieve the leaf index pages.

Example 4.5 (Index Only Queries)

If we look at the following query

$$\pi_A(\sigma_{A>v}(R)) \tag{41}$$

we see that we only care about the value of A and not the rest of the tuples, so we only need to look at the index pages and not the data itself.

Example 4.6 (Primary Index Clustered According to Search Key)

If we have a primary index, in most cases the actual records are also stored in the index pages/leaf nodes. If they are clustered according to attribute A, then one lookup can lead to all result tuples in their entirety. You can just hit a leaf and grab your records as you walk along the leaves.

Example 4.7 (Other Index-Only Queries)

For example, if we just wanted to look at the count of users with age 50, then we don't need the data. We can just look at the number of values in the leaf nodes of the B+ tree with this value.

```sql
SELECT E.do COUNT(*)
FROM Emp E
GROUP BY E.dno;
```

If we have an index on `(E.dno, E.sal)`, then the two queries are also index-only plans. However, if we index on `(E.dno)`, then we need to retrieve `E.sal` on the data page, incurring more cost.

```sql
SELECT E.dno, MIN(E.sal)
FROM Emp E
GROUP BY E.dno;
```

```sql
SELECT AVG(E.sal)
FROM Emp E
GROUP BY E.dno;
```

Example 4.8 (Halloween Problem)

The Halloween problem refers to a phenomenon in databases where an update operation causes a change in the physical location of a row, potentially allowing the row to be visited again later in the same update operation. Look at the update.

```
UPDATE Payroll
SET salary = salary * 1.1
WHERE salary <= 25000;
```

This caused everyone to have a salary of 25000+. This is because when we updated someone with salary of say 1000, it went to 1100 and is moved further right in the B+ tree. Therefore, this is revisited again is increased again in the same update. To fix this, we could update the values in reverse, from the rightmost leaf node to the leftmost one so that increasing values are visited once. Or we can just create a to-do/done list that keeps track of which ones have been updated.

4.6 Exercises

Now let's go through a bunch of questions to clarify some of the IO cost computation.

Example 4.9

Assume that we have the query `SELECT * FROM User WHERE age = 50;` with the following assumptions:

1. Assume 12 Users with `age = 50`.
2. Assume one data page (block) can hold 4 User tuples (so $f = 5$).
3. Suppose searching for data entry requires 3 IOs in a B+ tree, which contain pointers to the data records (so $h = 3$).

If the index is clustered, then we can just traverse down the B+ tree to get to the leaf node containing the value 50.

1. We have a cost of 3 to traverse down the tree to access the index pages which show the memory addresses of the tuples on disk.
2. Then, we will find entries and we want to find the cost to access the data pages. There are 12 Users, and for every address, we load the entire block in memory, which will retrieve the other users of age 50 (since this is clustered). At best, we will retrieve 3 blocks (of 4 tuples each) and at worst, due to block overlap, we will retrieve 4 blocks and read the rest from memory. This gives us a cost of +3 or +4.

The total cost is 6. If the index is unclustered, then we are not guaranteed that the data with values 50 will be contiguous, so we will in the worst case have to look at 12 different blocks, leading to a total cost of $3 + 12 = 15$.

Example 4.10 (Index Problem)

Consider a table `Orders(OrderID, CustomerID, OrderDate, TotalAmount)` with 5,000,000 records stored in 100,000 disk blocks (=pages or units of I/O). The rows are not sorted on any particular attribute. There is a **clustered** B+ tree index on OrderID (the primary key), and an **unclustered** B+ tree index on CustomerID.

Assume the following:

- Each node in the B+ trees corresponds to one disk block.
- The OrderID B+ tree has 4 levels (including the root) and 10,000 leaf nodes.
- The CustomerID B+ tree has 5 levels (including the root) and 50,000 leaf nodes.
- You do not have enough space in memory to hold all data pages.
- The root nodes of both B+ trees are always kept in memory.

- The non-root index nodes and all data pages are initially on disk.
- All the leaves have pointers to the next left node both on the left and the right side.
- All data pages are fully utilized.
- All nodes in B+ trees are also fully utilized.
- Uniformity in all places. You can ignore page boundaries.

Question 1.1

How many disk I/Os are required (index and data) to retrieve all order records for a specific CustomerID using the index on the CustomerID? Assume there are 100 orders for this customer.

We must access 4 levels, with 5 levels in tree minus 1 for head already in memory (+4). There is only 1 leaf index pages containing the matching entries, since there are 5m records and 50k leaf nodes, meaning that each leaf node can contain 100 addresses at most when it is a dense index. Given that there are 100 matches, we just need to use this leaf node to access the disk. This is not clustered, so we must look through all 100 addresses, reading each block from disk for an addition cost of 100 IOs. Total is **104 IOs**.

Question 1.2

Suppose the Orders table is frequently queried for recent orders, and the performance is critical. The current clustered index on OrderID is not providing optimal performance for these queries. You are considering reorganizing the Orders table to cluster it on OrderDate instead.

Assume that orders are uniformly distributed over time. Everything else is the same as in the original question description **(i.e., the number of levels of the new B+ tree indexed on OrderDate is still 4 and the root is still in memory. And the number of leaf nodes is still 10,000)**. Ignore page boundaries.

Consider the following query:

```
SELECT * FROM Orders
WHERE OrderDate BETWEEN '2023-09-01' AND '2023-09-30';
```

How many disk I/Os are required in the worst case for the following query before and after the change? What is the impact of the change? Assume that 0.5% of the orders were made in September 2023.

For before, since the data is distributed uniformly, we must in the worst case check all data pages of the relation on disk.

1. We first have 3 IOs to go down to the leaf (4 levels minus 1 head already in memory).
2. We scan through all 10,000 leaves, but we are already on leaf 1, so need to traverse 9999 pointers to load each index leaf.

Therefore, the total IO for traversal is **10,002 IOs**. If we include the IOs for loading data pages, we also incur more costs. There are 100,000 disk blocks, so as we load all of them we incur an additional IO cost of 100,000, giving us **110,002 IOs**.

For after, if we cluster on OrderDate, then out of the 100,000 disk blocks, we assume that 0.5% of them, or 500 of them, will contain the relevant dates. Additionally, we assume that by uniformity, 0.5% of the 10,000 leaf nodes will contain these relevant addresses.

1. We traverse down to the leaf containing the address of the tuple with date attribute `2023-09-01`. This is 3 IOs.
2. We must traverse through the rest of the leaf nodes. Since we are already at the first one, we must load in an additional 49 leaf blocks, so 49 IOs.
3. As we load in each block, we are loading in a total of 500 data blocks from disk, so an additional 500 IOs.

We end up with **552 IOs**. If we exclude the IOs for loading the data pages, we incur fewer cost of **52 IOs** since this is only for traversal of the B+ tree.

The difference in these costs is just the first value minus the second value. For example, I can do

$$10,002 - 552 = 9450 \tag{42}$$

reduced IO costs.

Question 1.3

(a) How many disk I/Os are required in the worst case to insert a new order with order ID 2000 and customer ID 200 if updating the clustered index on the Order ID? (i.e., what is the I/O required for updating the index and inserting data records?)

(b) What would the result be if we updated the unclustered index on the CustomerID?

Assume all memory blocks are being used for this update only. Then we do not throw away an index page once it is read.

For (a),

1. You should first get 3 IOs to traverse down to the leaf.
2. Then you load the relevant data page from disk onto memory (1 IO).
3. Since this block is full and you want to add a tuple to it, you must split it into two pages. So you initialize two buffers in memory and write two separate blocks. Then you must write these blocks back into disk (2 IOs).
4. Then you split the leaf (assuming it's already in memory) by writing its 2 splits back, again by splitting it in memory (2 IOs). You this for the next node (since in worst case it's full), for a total of 3 more times as you traverse up the tree (6 IOs).
5. You finally want to write a new root into the disk, so you make one in memory and write it (1 IO).

This gives a total of **15 IOs**.

For (b), should be 17 or 18?

1. You should first traverse down to the leaf (+4 IOs).
2. Then you take a data page and write it in memory (+1 IO) since you don't need to split it to keep the clustering. Then you flush the new page to disk (+2 IOs).
3. Then you split the leaf node by writing its 2 splits back (+2). You do this 4 more times (+8 IOs).
4. You then want to write a new root node (+1). So in total +16 IOs.

5 Query Processing and Optimization

Definition 5.1 (Disk/Memory Blocks)

To set up some notation, let $B(R)$ represent the number of disk blocks that relation R takes up, and say M is the number of blocks available in our memory. We start off with some simple operations.

Example 5.1 (Basic Block Storage)

Suppose we have relation R with $|R| = 1000$ and each block can hold 30 tuples. Then $B(R) = \lceil 1000/30 \rceil = 34$, or 35 if there is overlap.

Definition 5.2 (Buffer Blocks)

It turns out that whenever we output data to stdout, the blocks need to be stored in a **buffer block**. If the buffer block is full, then it is flushed to stdout.

Example 5.2 (Cost of Querying Everything)

If we do `SELECT * FROM R`, then

1. we are at most retrieving $B(R)$ pages from disk to memory, so our IO cost is $B(R)$.
2. Our memory cost is 2 pages since we take each page, load it to memory, and put the answer in the output page. The next page (if any) can overwrite the previously loaded page.

We can stop early if we lookup by key. Remember that this is not counting the cost of writing the result out.

It turns out that the efficiency of most operations defined on modern DBMS depends on two things: sorting and hashing. We'll divide up our analysis this way, focusing on their applications in joining, which tends to be the mostly used and expensive operation. Note that there are many ways to process the same query. We can in the most basic sense just scan the entire relation in disk. We can sort it. We can use a tree or hash index, and so on. All have different performance characteristics and make different assumptions about the data, so the choice is really problem-dependent. What a DBMS does is implements all alternatives and let the *query optimizer* choose at runtime. We'll talk about the algorithms now and talk about the query optimizer later.

5.1 Brute-Force Algorithms

We start off with the most brute-force algorithms of theta joins. In here, we describe how to implement it, its IO cost, and its memory cost. In the following, when we compute $R \bowtie_p S$, R is called the **outer table** and S the **inner table**. Furthermore, another convention is that when we calculate IO cost, we *do not* factor in the cost of writing our result back to disk.

5.1.1 Nested Loop Joins

Our first algorithm simply takes in every block from R, and for every tuple $r \in R$, we take in every tuple in S to calculate the predicate $p(r, s)$.

Algorithm 5.0 (Nested Loop Join)

Nested-loop join just uses a brute-force nested loop when computing $R \bowtie_p S$.

Algorithm 1 Nested loop Join

```
Require: Outer table $R$, Inner table $S$
  function NestedLoop(R, S)
    for each block $B_R \in R$ do
      load $B_R$ into memory block $M_1$
      for each tuple $r \in B_R$ do
        for each block $B_S \in S$ do
          load $B_S$ into memory block $M_2$
          for each tuple $s \in B_S$ do
            if $p(r, s)$ is true then
              Write $r \bowtie s$ into buffer block $M_3$
              Flush $M_3$ to stdout when it's full.
            end if
          end for
        end for
      end for
    end for
  end function
```

The IO cost of this is

1. $B(R)$ to load R
2. For every tuple $r \in R$, we run through all of $s \in S$, requiring $B(S)$ IOs.

Therefore

$$\text{IO} = B(R) + |R| \cdot B(S) \tag{43}$$

with memory cost 3 (since we use M_1, M_2, M_3 blocks).

This is clearly not efficient since it requires a lot of IOs. We can make a slight improvement by trying to do as much as we can with the loaded blocks in memory.

Algorithm 5.0 (Block-Based Nested-Loop Join)

Block-based nested-loop join loops over the blocks rather than the tuples in the outer loops.

```
Require: Outer table $R$, Inner table $S$
  function BlockNestedLoopJoin($R$, $S$)
    for each block $B_R \in R$ do
      load $B_R$ into memory block $M_1$
      for each block $B_S \in S$ do
        load $B_S$ into memory block $M_2$
        for each $r \in B_R, s \in B_S$ do
          if $p(r, s)$ is true then
            Write $r \bowtie s$ into buffer block $M_3$
            Flush $M_3$ to stdout when full.
          end if
        end for
      end for
    end for
  end function
```

The IO cost of this is

1. $B(R)$ to load R.
2. For every block B_R, we run through all of $s \in S$, requiring $B(S)$ IOs.

Therefore

$$\text{IO} = B(R) + B(R) \cdot B(S) \tag{44}$$

The memory cost is 3 (since we use M_1, M_2, M_3 blocks).

Algorithm 5.0 (Saturated Block-Based Nested-Loop Join)

The next optimization is to use more memory by basically stuffing the memory with as much of R as possible, stream S by, and join every S tuple with all R tuples in memory.

```
Require: Outer table $R$, Inner table $S$
  function SatBlockNestedLoopJoin($R$, $S$)
    for each set of blocks $\mathbf{B} = \{B_{i+1}, \ldots, B_{i+(M-2)}\} \subset R$ do
      load $\mathbf{B}$ into memory blocks $M_1, \ldots, M_{M-2}$.
      for each $B_S \in S$ do
        load $B_S$ into memory block $M_{M-1}$.
        for each $r \in \mathbf{B}, s \in B_S$ do
          if $p(r, s)$ is true then
            Write $r \bowtie s$ into buffer block $M_M$.
            Flush $M_M$ to stdout when it's full.
          end if
        end for
      end for
    end for
  end function
```

The total IO cost of this is

1. $B(R)$ to load R.
2. For every set of $M - 2$ blocks, we run through all of $s \in S$, requiring $B(S)$ IOs.

Therefore

$$B(R) + \left\lceil \frac{B(R)}{M-2} \right\rceil \cdot B(S) \approx B(R) \cdot B(S)/M \tag{45}$$

You want to pick the bigger table as R since you want the smaller table S to be loaded/streamed in multiple times.

Algorithm 5.0 (Index Nested Loop Join)

If we want to compute $R \bowtie_{R.A=S.B} S$, the idea is to use the value of $R.A$ to probe the index on $S(B)$. That is, for each block of R, we load it into memory, and for each r in the block, we use the index on $S(B)$ to retrieve s with $s.B = r.A$, and output rs.
The IO runtime, assuming that S is unclustered and secondary, is

$$B(R) + |R| \cdot (\text{indexlookup}) \tag{46}$$

Since typically the cost of an index lookup is 2-4 IOs, it beats other join methods mentioned later if $|R|$ is not too big. Since this does not scale at all with S, it is better to pick R to be the smaller relation. The memory requirement as with other operations is 3 blocks.

5.2 Sort-Based Algorithms

5.2.1 External Merge Sort

Now let's talk about processing of queries, namely how sorting works in a database system, called **external merge sort**. In an algorithm course, we know that the runtime is $O(n \log n)$, but this is for CPU comparisons where the entire list is loaded in memory. This is extremely trivial in comparison to the IO commands we use in databases, so we will compute the runtime of sorting a relation by an attribute in terms of IO executions.

The problem is that we want to sort R but R does not fit in memory. We divide this algorithm into **passes** which deals with intermediate sequences of sorted blocks called **runs**.

1. Pass 0: Read M blocks of R at a time, sort them in memory, and write out the sorted blocks, which are called *level-0 runs*.
2. Pass 1: Read $M - 1$ blocks of the level 0 runs at a time, sort/merge them in memory, and write out the sorted blocks, which are called *level-1 runs*.[4]
3. Pass 2: Read $M - 1$ blocks of the level 1 runs at a time, sort/merge them in memory, and write out the sorted blocks, which are called *level-2 runs*.
4. ...
5. Final pass produces one sorted run.

[4] The reason we need $M - 1$ rather than M is that now we are merging. We are not merging in-place so we need this extra buffer to store as we traverse our pointers down each of the $M - 1$ blocks.

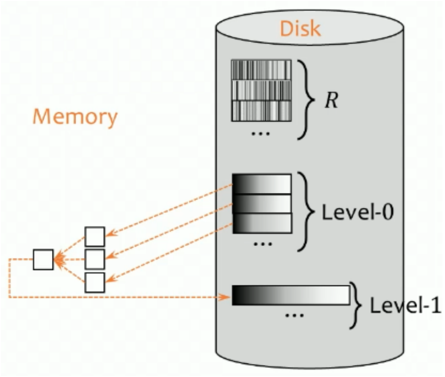


Figure 39

Algorithm 5.0 (External Merge Sort)

The implementation has a lot of details. Not finished yet.

```
Require: Relation $R$
  function ExternalMergeSort($R$)
      $L = [0]$                                              ▷Array storing number of level-$i$ runs
      while while there are blocks to read from $R$ do                          ▷First pass
          read the next $M$ blocks $B = \{B_1, \ldots, B_M\}$ at a time and store it in memory.
          sort $B$ to generate a level-0 run $B^{(0)} = \{B'_1, \ldots, B'_M\}$.
          write $B^{(0)}$ to disk.
          $L[0] += 1$
      end while
      while $L[-1] \geq M$ do
          append 0 to $L$
          let $\mathbf{B} = \{B\}$ be the set of previous runs
          while there exists blocks to be read in $\mathbf{B}$ do
              read $M - 1$ blocks starting from the beginning of each run into memory.
              sort them to produce the $i$th run $B^{(i)}$.
              write $B^{(i)}$ to disk.
          end while
      end while
  end function
```

To compute the cost, we know that

1. in pass 0, we read M blocks of R at a time, sort them, and write out a level 0 run, so there are
$$\lceil B(R)/M \rceil \tag{47}$$
level 0 sorted runs, or passes.
2. in pass i, we merge $M - 1$ level $(i - 1)$ runs at a time, and write out a level i run. We have $M - 1$ memory blocks for input and 1 to buffer output, so
$$\text{Num. of level i runs} = \left\lceil \frac{\text{Num. of level (i-1) runs}}{M - 1} \right\rceil \tag{48}$$
3. The final pass produces 1 sorted run.

Therefore, the number of passes is approximately
$$\left\lceil \log_{M-1} \left\lceil \frac{B(R)}{M} \right\rceil \right\rceil + 1 \tag{49}$$

and the number of IOs is $2B(R)$ since each pass reads the entire relation once and write it once. The memory requirement is M (as much as possible).

Example 5.3 (Baby Merge)

Assume $M = 3$, with each block able to hold at most 1 number. Assume that we have an input (relation)

$$1, 7, 4, 5, 2, 8, 3, 6, 9 \tag{50}$$

Then we go through multiple passes.

1. Pass 0 will consist of 3 runs. You load each of the 3 numbers in memory and sort them.

$$1, 7, 4 \mapsto 1, 4, 7 \tag{51}$$
$$5, 2, 8 \mapsto 2, 5, 8 \tag{52}$$
$$9, 6, 3 \mapsto 3, 6, 9 \tag{53}$$

2. Pass 1. You merge them together by first taking 1 and 2, loading them in memory, and then comparing which one should go first. Once 1 is outputted, then the next number 4 overwrites 1 in memory, and then 2 is outputted, and so on.

$$1, 4, 7 + 2, 5, 8 \mapsto 1, 2, 4, 5, 7, 8 \tag{54}$$
$$3, 6, 9 \tag{55}$$

3. Pass 2. Merges the final two relations.

$$1, 2, 3, 4, 5, 7, 8 + 3, 6, 9 \mapsto 1, 2, 3, 4, 5, 6, 7, 8, 9 \tag{56}$$

Therefore, pass 0 uses all M pages to sort, and after that, when we merge, we only use $M - 1$ pages to merge the inputs together and 1 page for the output.

Some performance improvements include:

1. *Double Buffering.* You allocate an additional block for each run, and while you are processing (merging the relations in memory), you run the IO concurrently and store it in the new block to save some time.
2. *Blocked IO.* Instead of reading/writing one disk block at a time, we can read/write a bunch of them in clusters. This is sort of like parallelization where you don't output just one block, but multiple blocks done from multiple processing.

The problem with both of these is that we have smaller fan-in, i.e. more passes. Since we are using more blocks per run than we have, we can look at fewer runs at once.

5.2.2 Sort Merge Joins

Now that we know how to sort, let's exploit this to optimize joins beyond nested-loops. We introduce a naive version of sort-merge join.

Algorithm 5.0 (Naive Sort-Merge Join)

A clever way is to first sort R and S by their join attributes, and then merge. Given that the first tuples in sorted R, S is r, s, we do repeat until one of the R or S is exhausted.

1. If $r.A > s.B$, then $s =$ next tuple in S
2. Else if $r.A < s.B$, then $r =$ next tuple in R.
3. Else output all matching tuples and $r, s =$ next in R, S, which is basically a nested loop.

Therefore, given that it takes $\text{sort}(R), \text{sort}(S)$ to sort R, S (the equations above is too cluttered to write), the IO cost consists of

1. Sorting R and S.
2. We then must write R and S to disk in order to prepare for merging, so $B(R) + B(S)$.
3. We then must write R and S back into memory, one at a time, to merge them, so $B(R) + B(S)$.

Therefore, the IO is really just $2B(R) + 2B(S)$ more than it takes to sort both R and S.

$$\text{IO} = \text{sort}(R) + \text{sort}(S) + 2B(R) + 2B(S) \tag{57}$$

which is worst case $B(R) \cdot B(S)$ when everything joins.

Algorithm 2

```
Require:

  function FUNC(x)
  end function
```

Example 5.4 (Worst Case)

To see the worst case when the IO cost is $B(R) \cdot B(S)$, look at the following example. By the time we got to the first 3, we can't just increment the pointers for both relations. We must scan through all of the tuples of A with value 3 and all those in B with value 3 and do a nested loop to join them.

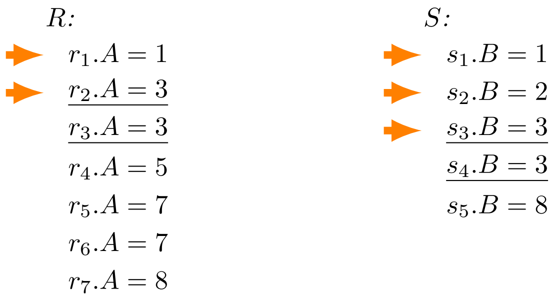


Figure 40: Before you increment the next pointer, you must loop through all combinations of the underlined elements in the left and right relations.

We have completely isolated the sorting phase and the merging phase, but we don't have to do this. Just like regular merge-sort, we can integrate them together to save IO in the last merge phase.

Algorithm 5.0 (Optimized Sort-Merge Join)

The algorithm is just slightly modified from the naive implementation. After the final pass from sorting both R and S, say that we have W_R and W_S final runs such that

$$M > W_R + W_S \tag{58}$$

We can assume that this is true since if it wasn't, we can just add another pass of external sort to reduce one of the W's. Then, we can do these 3 things simultaneously.

1. We can load in the next smallest block from R from its runs, merge them together.
2. We can load in the next smallest block from S form its runs, merge them together.
3. We can join the merged blocks *in memory* and output to the buffer to be flushed.

This saves us the IO cost of writing the sorted runs back into memory and then loading them again to write, giving us a total IO cost that is equal to that of simply sorting R and S.

$$\text{IO} = \text{sort}(R) + \text{sort}(S) \tag{59}$$

The memory varies depending on how many passes, but if R and S are moderately big in that we need full memory to sort them, then the memory cost is M (we use everything).

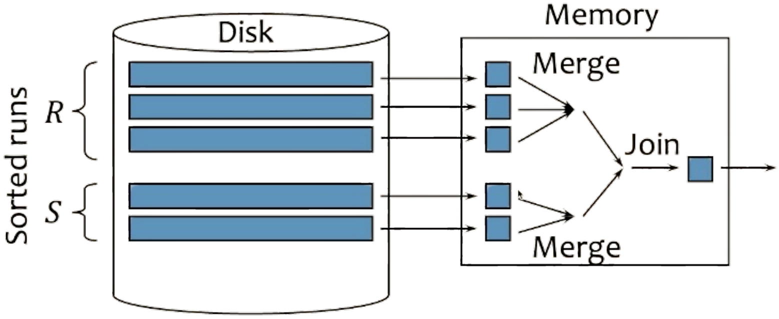


Figure 41: Sorting: produce sorted runs for R and S such that there are fewer than M of them total. Then in merge and join, we merge the runs of R, merge the runs of S, and merge-join the result streams as they are generated!

Example 5.5 (Two-Pass SMJ)

If SMJ completes in two passes, then the IOs is really cheap since we are basically getting a $2B(R) + 2B(S)$ cost of a level-0 pass, plus the final merge-join step which takes another $B(R) + B(S)$.

$$\text{IO} = 3(B(R) + B(S)) \tag{60}$$

If SMJ cannot complete in 2 passes, then we repeatedly merge to reduce the number of runs as necessary before the final merge and join.

5.2.3 Zig-Zag Join

Definition 5.3 (Zig Zag Join using Ordered Indices)

To compute $R \bowtie_{R.A=S.B} S$, the idea is to use the ordering provided by the indices on $R(A)$ and $S(B)$ to eliminate the sorting step of merge-join. The idea is similar to sort-merge join. We start at the leftmost leaf node of both indices of R and S, and traverse (right) through the leaves, querying both of the data at leaf a in R and b in S if the leaf values are equal.
Note that we don't even have to traverse through all leaves. If we find that a key is large, we can just start from the root node to traverse, possibly skipping many keys that don't match and not incurring all those IO costs. This can be helpful if the matching keys are distributed sparsely.

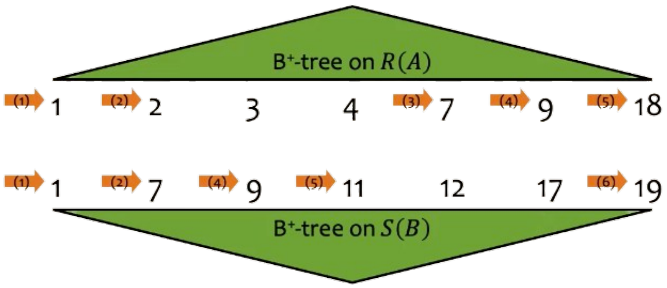


Figure 42: We see that the B+ tree of B has value 7 while that of A has value 2. Rather than traversing $2 \mapsto 3 \mapsto 4 \mapsto 7$, we can just traverse to 7 from the root node of A. This can give us a stronger bound.

5.2.4 Other Sort Based Algorithms

The set union, intersection, difference is pretty much just like SMJ.

For duplicate elimination, you simply modify it so that during both the sort and merge steps, you eliminate duplicates if you find any.

For grouping and aggregation, you do external merge sort by the group-by columns. The trick is you produce "partial" aggregate values in each run and combine them using merge.

5.3 Hash-Based Algorithms

5.3.1 Hash Join

Hash joining is useful when dealing with equality predicates: $R \bowtie_{R.A=S.B} S$.

Definition 5.4 (Hash Join)

The main idea of **hash join** is that we want to partition R and S by hashing (using a hash function that maps to $M-1$ values) their join attributes and then consider corresponding partitions (that get mapped to the same hash value) of R and S. If $r.A$ and $s.B$ get hashed to the same number, they might join, and if they don't, then they definitely won't join. The figure below nicely visualizes this.

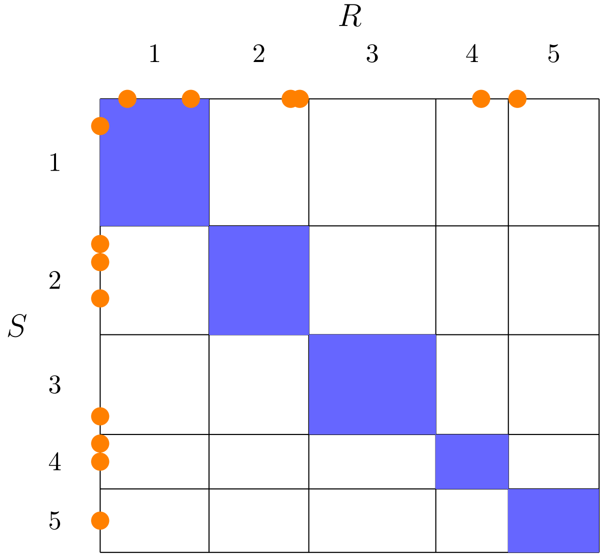


Figure 43: Say that the orange points represent the hashed values of $r.A$ and $s.B$ for $r, s \in R, S$. The nested loop considers all slots (all pairs of orange points between R and S), but hash join considers only those along the diagonal.

Then, in the **probing phase**, we simply read in each partition (the set of tuples that map to the same hash) of R into memory, stream in the corresponding partition of S, compare their *values* (not hashes since they are equal), and join if they are equal. If we cannot fit in a partition into memory, we just take a second hash function and hash it again to divide it into even smaller partitions (parallels merge-sort join).

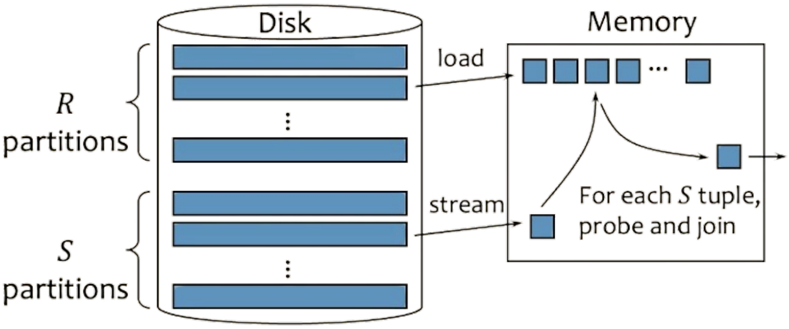


Figure 44

Therefore, if the hash join completes in two passes, the IO runtime is

$$3(B(R) + B(S)) \tag{61}$$

which is similar to merge-sort join. As for the memory requirement, let's first assume that in the probing phase, we should have enough memory to fit one partition of R, i.e. $M - 1 > \lceil B(R)/(M-1) \rceil$, so solving for it roughly gives $M > \sqrt{B(R)} + 1$. We can always pick R to be the smaller relation, so roughly

$$M > \sqrt{\min\{B(R), B(S)\}} + 1 \tag{62}$$

Theorem 5.1 (Hash vs Sort-Merge Join)

To compare hash join and SMU, note that their IOs are the same, but for memory requirements, hash join is lower, especially when the two relations have very different sizes.

$$\sqrt{\min\{B(R), B(S)\}} + 1 < \sqrt{B(R) + B(S)} \tag{63}$$

Some other factors include the quality of the hash (may not generate evenly sized partitions). Furthermore, hash join does not support inequality joins unlike SMJ, and SMJ wins if either R and/or S is already sorted. SMJ also wins if the result needs to be in sorted order.

Sometimes, even block nested loop join may win in the following cases.

1. if many tuples join (as in the size of the join is $|S| \cdot |R|$) since we are doing unnecessary processing in hash/merge-sort joins.
2. if we have black-box predicates where we may not know the truth/false values of the θ

5.3.2 Other Hash Based Algorithms

The union, difference, and intersection are more or less like hash join.

For duplicate elimination, we can check for duplicates within each partition/bucket.

For grouping/aggregation, we can apply the hash functions to the group-by columns. Tuples in the same group must end up in the same partition/bucket. Or we may keep a running aggregate value for each group.

To compare the duality of sort and hash, note that

1. in sorting, we have a physical division and logical combination
2. in hashing, we have a logical division and physical combination

When handling large inputs,

1. in sorting we have multi-level merge
2. in hashing we have recursive partitioning

For IO patterns,

1. in sorting we have sequential write and random read (merge)
2. in hashing we have random write and sequential read (partition)

5.4 Exercises

Here are some exercises for calculating IO costs of these joins.

Example 5.6 (Join Operations)

Consider the two tables

1. `Orders(OrderID, CustomerID, OrderDate, TotalAmount)`
2. `Customers(CustomerID, Address)`

`Orders.CustomerID` is a foreign key referring to `Customers.CustomerID`.
The rows are not sorted on any particular attribute. We want to join the two tables using the attribute CustomerID:

$$\text{Orders} \bowtie_{\text{Orders.CustomerID} \, = \, \text{Customers.CustomerID}} \text{Customers}$$

The inner and outer tables can be swapped to reduce I/O costs in the following questions.
Assume the following:

- The cost metric is the number of page/block I/Os unless otherwise noted.
- DO NOT count the I/O cost of writing out the final result unless otherwise noted.
- M = 19 blocks (= pages) available in memory unless otherwise noted.
- Table Orders contains 50,000 records (rows/tuples) on disk. One block can contain 20 Orders-tuples.
- Table Customers contains 20,000 records (rows/tuples) on disk. One block can contain 10 Customers-tuples.
- Assume uniform distribution for Orders.CustomerID and Customers.CustomerID – i.e. the same number of Orders-tuples join with a Customers-tuple.
- Ignore page boundary wherever applicable.
- Assume uniform distributions wherever applicable.

Question 2.1

(a) For the Orders and Customers tables, we want to perform a nested-loop join (**using 3 memory blocks**). What is the **minimum** total I/O cost? (Choose the table that will reduce the I/O cost as the outer table.)

(b) For the Orders and Customers tables, we want to perform a block-based nested-loop join (**using 3 memory blocks**). What is the **minimum** total I/O cost?

For (a), we choose $R =$ Customers as the outer table since it is smaller. Therefore, our IO cost is

$$B(R) + |R| \cdot B(S) = 2000 + 20,000 \cdot 2500 = 50,002,000 \tag{64}$$

For (b), we also choose $R =$ Customers as outer since it's smaller. The IO cost is

$$B(R) + B(R) \cdot B(S) = 2000 + 2000 * 2500 = 5,002,000 \tag{65}$$

Question 2.2

(a) If we want to perform an external merge sort on the Orders table, how many level-0 runs does the external merge sort produce for the Orders table (M=19)?

(b) Continuing with the question (a), how many passes in total does the external merge sort take (including the first sorting pass)? Show the calculations for each pass (including the number of runs and size of each run).

(c) What is the total I/O cost of the external merge sort for the Orders table? Remember, do not count the cost of final write.

(d) Do an improved sort-merge-join, i.e., do merge and join in the same pass when the total number of sorted runs from Orders and Customers will fit in memory including an output block. Compute the minimum total I/O cost of sort-merge-join of table Orders and Customers.

For (a), we read M blocks of R at a time, so we need to have $\lceil B(R)/M \rceil = \lceil 2500/19 \rceil =$ **132 passes**.

For (b), we saw that

1. The level 0 pass takes 132 runs.
2. The level 1 pass takes $\lceil 132/(19-1) \rceil = 8$ runs.
3. At this point we can just run once more since $8 < 18$, so our level 2 pass takes 1 run.

So we have **3 passes** with **141 runs**.

For (c), we have $B(R) = 2500$ and just compute the following.

1. For level 0, we read all blocks, and write them all out: $2B(R)$.
2. For level 1, it's the same thing since we again read all blocks and write them back to disk: $2B(R)$.
3. For level 2, we just read through and do not include the final write, so $B(R)$.

This is a total of $5B(R) = 5 \cdot 2500 =$ **12,500 IOs**.

For (d),

1. For the first pass, you load all blocks of both relations in memory for the first iteration of sort merge and then write them to disk, giving us $2(B(R) + B(S))$ IOs.
2. The second pass is the same, giving us $2(B(R) + B(S))$ IOs.
3. By the third pass, we complete the sort-merge join giving us $B(R) + B(S)$.

This is a total of $5(B(R) + B(S)) = 5 \cdot (2000 + 2500) =$ **22,500 IOs**.

Question 2.3 Assume uniform distribution for the hash function(s).

(a) For the Orders and Customers tables, we want to perform a multi-pass hash join. How many passes do we need (including partitioning phase and join phase)?
(b) What is the minimum I/O cost of joining Orders and Customers using a hash join?
(c) What is the minimum number of memory blocks required if we want to perform a 2-pass hash join? Note that M should be an integer.

For (a), we must partition both relations.

1. In the first partition pass, we
   (a) take Customers and have $\lceil 2000/(19-1) \rceil = 112 > 18$. Each partition contains 112 blocks, which is too big for our memory, so we must partition again.
   (b) take Orders and have $\lceil 2500/(19-1) \rceil = 139$. Each partition contains 139 blocks, which is too big for our memory, so we must partition again.
2. In the second partition pass, we
   (a) take Customers and have $\lceil 112/18 \rceil = 7 \leq 18$, so we are done since each partition (of a partition) of 7 blocks can fit in memory.
   (b) take Orders and have $\lceil 139/18 \rceil = 8 \leq 18$, so we are done since each partition (of a partition) of 8 blocks can fit in memory.
3. Finally, we load each partition of Customers in memory (7 blocks), and for each partition, we just iterate through the corresponding partition of Customers. This is one join pass.

There are a total of **3 passes**, or 5 passes if we consider each partition step of each relation as an individual pass.

For (b), we see that since `Orders.CustomerID` is a foreign key, we won't have a case where every row in `Orders` will trivially join with every row in `Customers`. We can compute the steps as such.

1. The 1st partition pass.
   (a) For Customers, it requires you to read all blocks (2000 IOs) and then write $112 \cdot 18$ blocks back onto disk (2016 IOs),[a] or we are just approximating, this is about 2000 IOs as well.
   (b) For Orders, it requires you to read all blocks (2500 IOs) and then write $139 \cdot 18$ blocks back onto disk (2502 IOs), or just 2500 IOs as an approximation. This is flushed out as well.

   This gives us a total of $2000 + 2016 + 2500 + 2502 = 9018$ IOs or with the approximations 9000 IOs.
2. The 2nd partition pass.
   (a) For Customers, now you have 18 partitions with each 112 blocks. For each partition, you load it again (112 IOs) and then partition it again to write back $18 \cdot 7$ blocks (126 IOs) for a total of 238 IOs. You do this 18 times for each partition giving us $238 \cdot 18 = 4284$ IOs. Again, we can just approximate it by saying that loading all partitions is 2000 and writing all is 2000, giving us 4000 IOs.
   (b) For Orders, now you have 18 partitions with each 139 blocks. For each partition, you load it again (139 IOs) and then partition it again to write back $18 \cdot 8$ blocks (144 IOs) for a total of 283 IOs. You do this 18 times for each partition giving us $283 \cdot 18 = 5904$ IOs. Again, we can just approximate it by saying that loading all partitions is 2500 and writing all is 2500, giving us 5000 IOs.

   This gives us a total of $4284 + 5904 = 10188$ IOs, or with the approximations 9000 IOs.
3. In the join phase, you load the partitions of R and S with the matching hashes once each to compare, so we have (not including write)
   (a) the partitions of R, which is $18 \cdot 18 \cdot 7 = 2268$ IOs, or 2000 IOs approximately.
   (b) the partitions of S, which is $18 \cdot 18 \cdot 8 = 2592$ IOs, or 2500 IOs approximately.

   giving us a total of 4860 IOs.

Therefore, we have a total of $9018 + 10188 + 4860 =$ **24066 IOs**, or if we use our approximations, it's simply $5(B(R) + B(S)) = 5 \cdot 4500 =$ **22,500 IOs**.

For (c), M must satisfy

$$M \geq \lceil \sqrt{\min\{2500, 2000\}} \rceil + 1 = 46 \tag{66}$$

giving us $M = 46$. Checking this is indeed the case, since now the image of our hash function is $\{1, \ldots, 45\}$, and assuming uniformity the number of blocks in each partition is $\lceil 2000/45 \rceil = 45$, which is just enough to fit in memory and then use the last block as an input buffer for the bigger relation when joining.

[a] This is not exactly 2000 because $2000/18 = 111.11$, which means that after repeatedly flushing out the filled partitions to disk 111 times, at the end we will have a partially filled buffer block in memory for each of the 18 partitions. This will need to be flushed out as well.

5.5 Logical Plans

When the DBMS chooses the best way to sort or merge two relations, it needs to choose it immediately. This can be done crudely by looking at the statistics of the relations that it is working with, but it doesn't guarantee that you will get the optimal plan. Therefore, it goes by the principal that you shouldn't try and waste time choosing the optimal one, but rather avoid the horrible ones. As a user of this DBMS, we should also take care in writing queries that are not too computationally or IO heavy. Two general heuristics that we should follow are:

1. You want to *push down*, i.e. use as early as possible, selections and projections.
2. You want to join smaller relations first and avoid using cross product, which can be devastating in memory.

Definition 5.5 (Logical Plan)

To have an approximate sense of how computationally heavy a query is, we can construct a high-level **logical plan**, which shows the computation DAG of the relational operators that we will perform for a query. We can optimize the logical plan by modifying our intermediate steps, such as optimizing our relational algebra logic or our implementation of SQL.

5.5.1 Query Rewrite

Using what we know, we can get a bit more theoretical and use the following identities in relational algebra. However, this has already been optimized and only does so much in practice.

Theorem 5.2 (Identities)

The following hold:

1. Selection-Join Conversion: $\sigma_p(R \times S) = R \bowtie_p S$
2. Selection Merge/Split: $\sigma_{p_1}(\sigma_{p_2} R) = \sigma_{p_1 \wedge p_2} R$
3. Projection Merge/Split: $\pi_{L_1}(\pi_{L_2} R) = \pi_{L_1} R$, where $L_1 \subseteq L_2$
4. Selection Push Down/Pull Up: $\sigma_{p \wedge p_r \wedge p_s}(R \bowtie_{p'} S) = (\sigma_{p_r} R) \bowtie_{p \wedge p'} (\sigma_{p_s} S)$, where:
   (a) p_r is a predicate involving only R columns
   (b) p_s is a predicate involving only S columns
   (c) p and p' are predicates involving both R and S columns
5. Projection Push Down: $\pi_L(\sigma_p R) = \pi_L(\sigma_p(\pi_{LL'} R))$, where L' is the set of columns referenced by p that are not in L

Definition 5.6 (SQL Query Rewrite)

We can rewrite SQL queries directly, though this is more complicated and requires knowledge of the nuances of the DBMS.

1. Subqueries and views may not be efficient, as they divide a query into nested blocks. Processing

each block separately forces the DBMS to use join methods, which may not be optimal for the entire query though it may be optimal for each block.

2. Unnest queries convert subqueries/views to joins.

Therefore, it is usually easier to deal with select-project-join queries, where the rules of relational algebra can be cleanly applied.

Example 5.7 (Query Rewrite)

Given the query, we wish to rewrite it.

```
SELECT name
FROM User
WhERE uid = ANY(SELECT uid FROM Member);
```

The following is wrong since there may be one user in two groups, so it will be duplicated.[a]

```
SELECT name
FROM User, Member
WHERE User.uid = Member.uid;
```

The following is correct assuming `User.uid` is a key.

```
SELECT name
FROM (SELECT DISTINCT User.uid, name)
FROM User, Member
WHERE User.uid = Member.uid);
```

[a] A bit of review: when testing whether two queries are equal, think about if the two queries treat duplicates, null values, and empty relations in the same way.

Example 5.8 (Correlated Subqueries)

Look at this query where we want to select all group ids with name like Springfield and having less than some number of members.

```
SELECT gid
FROM Group, (SELECT gid, COUNT(*) AS cnt FROM Member GROUP BY gid) t
WHERE t.gid = Group.gid AND min_size > t.cnt
AND name LIKE ’Springfield%’;
```

This is inefficient since for every `gid`, we are making an entire extra query to select the counts. This is called a **non-correlated** query since this subquery is being run independently for every run. It ends up computing the size of *every* group, unlike the following one, where it filters out groups named Springfield first and then computes their size.

```
SELECT gid FROM Group
WHERE name LIKE ’Springfield%’
AND min_size > (SELECT COUNT(*) FROM Member WHERE Member.gid = Group.gid);
```

5.5.2 Search Strategies

Given a set of operations we have to do, the number of permutations that we can apply these operations grows super-exponentially. The problem of finding the best permutation is called a **search strategy**.

Example 5.9 (Left-Deep Plans)

Say that we have relations $R_1, \ldots, R_n$ that we want to join. The set of all sequences in which we can join them is bijective to the set of all binary trees with leaves R_i. This grows super-exponentially, reading $30,240$ for $n = 6$. There are too many logical plans to choose from, so we must reduce this search space. Here are some heuristics.

1. We consider only **left-deep** plans, in which case every time we join two relations, it is the outer relation in the next join.[a]

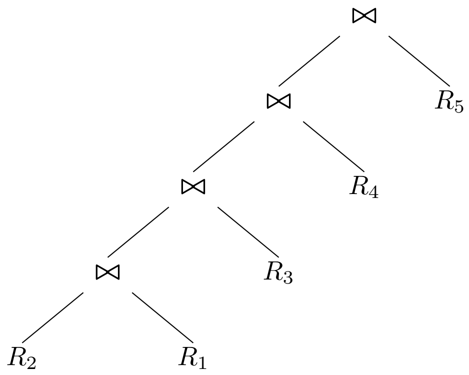


Figure 45: Left deep plans have a search space of only $n!$, which is better than before.

2. We can consider a balanced binary tree, which can be parallel processed, but this causes more runtime on the CPU in sort-merge joins, you must materialize the result in the disk, and finally the search space of binary trees may be larger than that of the left-deep tree.

[a] Note that since the right/inner relation is the one that is being scanned, we want the right one to be smaller since for each block of the left relation, we are looping over all blocks of the right relation. Therefore, left-deep plans are much more efficient since we don't have to scan the huge relation from the disk multiple times. We can just send it directly to the next join.

Even left-deep plans are still pretty bad, and so optimizing this requires a bit of DP (dynamic programming), using *Selinger's algorithm.*

Algorithm 5.0 (Selinger's Algorithm)

Given $R_1, \ldots, R_n$, we must choose a permutation from the $n!$ permutations. Say that the cost of the optimal join of a set $\mathbb{R}$ is $f(\mathbb{R})$. Note the recursive formula for some $S \subset [n]$.

$$f(\{R_i\}_{i \in S}) = \min_i f(\{R_j\}_{j \in S, j \neq i}) + f(R_i, \bowtie_{j \in S, j \neq i} R_j) \tag{67}$$

Where we sum up the cost of getting the accumulated relation and add it to the additional cost of joining once more with R_i. Therefore, given the R_i's,

1. We compute all $f(\{R_i, R_j\})$ for $i < j$ (since $j > i$ requires us the larger one to be inner).
2. Then we apply the recursive formula for all 3-combinations and so on, until we get to n-combinations.

Given a certain logical plan, the DBMS tries to choose an optimal physical plan as we will see later. However, the globally optimal plan may not be achieved with a greedy approach of first choosing the optimal logical plan and then its optimal physical plan. Due to the sheer size of the search space, we tend to go for "good" plans rather than optimal ones.

5.6 Physical Plan

The logical plan gives us an abstract view of the operations that we need to perform. It is mainly defined at the relational algebra or language level. But simply sorting or joining two relations is not done in just one way (e.g. we can scan, hash, or sort in different ways). Optimizing the logical plan may or may not help in the runtime, since operations are dependent on the size of the intermediate inputs.

Definition 5.7 (Physical Plan)

The actual sub-decisions needed to execute these operations constitute the **physical plan**, which is the actual implementation including even more operations in between each node of the logical plan. Here are a few terms to know.

1. *On the fly.* The computations are done in memory and are not written back to disk.
2. *(Index) Scan.* We scan for index value using a clustered/unclustered index on a B+ tree.
3. *Sort.* Usually means external merge sort.
4. *Filter.* Means the same as selection.

The difference between the logical and physical plan is that the logical plan represents *what* needs to be done (which we write SQL) and not *how* (which the DBMS chooses). Consider the two approaches.

Example 5.10 (Query Plans)

Consider the following SQL query:

```sql
SELECT Group.title
FROM User
JOIN Member ON User.uid = Member.uid
JOIN Group ON Member.gid = Group.gid
WHERE User.name = 'Bart';
```

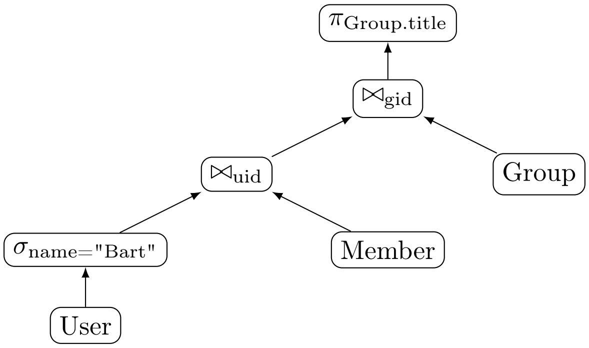


Figure 46: Logical Plan. We first take User, select Bart, and join it to Member over uid. Then we join it with Group on gid, and finally project the title attribute.

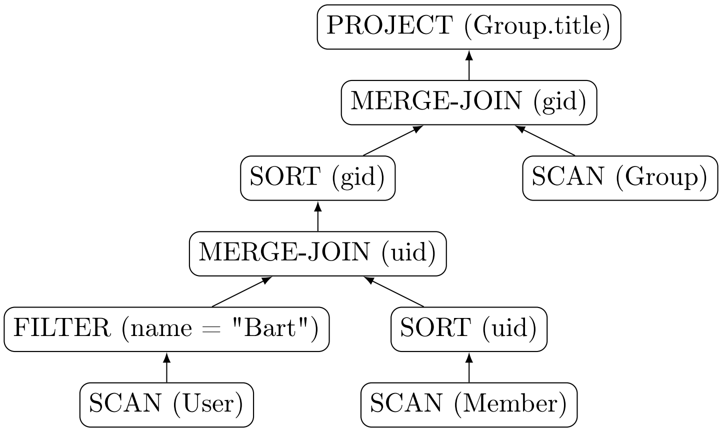


Figure 47: Physical Plan. We first take User and do a scan before filtering/selecting tuples with Bart. We also scan Member and sort it by uid in order to prepare for merge-join. Once we merge-join over uid, we sort it again to prepare a second merge-join with Group (which we scan first). Once we do this, we finally project the title attribute.

5.6.1 SQL Rewrite

At the language level, SQL provides some APIs to force the DBMS to use a certain physical plan if desired. This requires expertise and should not be done by beginners, however.

5.6.2 Cardinality Estimation

In the physical plan, we need to have a cost estimation for each operator. For example, we know that `SORT(gid)` takes $O(B(\text{input}) \cdot \log_M B(\text{input}))$, but we should find out what B, the number of blocks needed to store our input relation, is. To do this, we need the size of intermediate results through cardinality estimation.

Usually we cannot do quick and accurate cardinality estimation without strong assumptions, the first of which is uniformity of data.

Example 5.11 (Selection with Equality Predicates)

Suppose you have a relation R with $|R| = 100,000$ tuples. Assume that it has an attribute A taking integer values in $[50, 100)$ *distributed uniformly.* Then, there are 50 distinct values, and when we want to do $\sigma_{A=a}(R)$, then we would expect it to return

$$|\sigma_{A=a}(R)| = \frac{|R|}{|\pi_A(R)|} = 2000 \tag{68}$$

tuples.

The second assumption is *independence* of the distributions over each attribute.

Example 5.12 (Selection with Conjunctive Predicates)

If we have the same relation R with integer attributes $A \in [50, 100), B \in [10, 20)$ independently and uniformly distributed. Then,

$$|\sigma_{A=a,B=b}(R) = \frac{|R|}{|\pi_A(R)| \cdot |\pi_B(R)|} = \frac{100,000}{50 \cdot 10} = 200 \tag{69}$$

At this point, we are just using inclusion-exclusion principle and this becomes a counting problem.

Example 5.13 (Negated, Disjunctive Predicates)

We list these identities for brevity. The math is pretty simple.

$$|\sigma_{A \neq a}(R)| = |R| \cdot \left(1 - \frac{1}{|\pi_A(R)|}\right) \tag{70}$$

and using I/E principle, we have

$$|\sigma_{A=a \vee B=b}(R)| = |R| \cdot \left(\frac{1}{|\pi_A(R)|} + \frac{1}{|\pi_B(R)|} - \frac{1}{|\pi_A(R)| \cdot |\pi_B(R)|}\right) \tag{71}$$

Example 5.14 (Range Predicates)

Range also works similarly, but only if we know the actual bounds of the attribute values.

$$|\sigma_{A>a}(R)| = |R| \cdot \frac{\max(R.A) - a}{\max(R.A) - \min(R.A)} \tag{72}$$

Clearly, if we know that an attribute follows, say a Gaussian or a Poisson distribution, we can just calculate the difference in the CDFs and scale up by the relation size to get the approximate cardinality. I think this is what the professor refers to as *histogram estimation*.

For joins, we need yet another assumption, called *containment of value sets*. This means that if we are natural joining $R(A, B) \bowtie S(A, C)$, every tuple in the smaller (as in fewer distinct values for the join attribute A) joins with some tuple in the other relation. In other words,

$$|\pi_A(R)| \leq |\pi_A(S)| \implies \pi_A(R) \subset \pi_A(S) \tag{73}$$

which again is a very strong assumption in general but holds in the case of foreign key joins.

Example 5.15 (Two Way Equi-Join)

With the containment assumption, we have

$$|R \bowtie_A S| = \frac{|R| \cdot |S|}{\max(|\pi_A(R)|, |\pi_A(S)|)} \tag{74}$$

Think of this as looking at the cross product between the two relations, and then filtering out the actual tuples that don't belong there.

5.7 Exercises

Let's do a more comprehensive exercise.

Example 5.16 (Cost Estimation of Physical Query Plan)

Say we have three relations

1. `Student(sid, name, age, addr)`. $T(S) = 10,000$ tuples, $B(S) = 1,000$ pages.
2. `Book(bid, title, author)`. $T(B) = 50,000$ tuples, $B(S) = 5,000$ pages.
3. `Checkout(sid, bid, date)`. $T(C) = 300,000$ tuples, $B(C) = 15,000$ pages.

And say that the number of `author` attribute values in `Book` with $7 \leq$ `age` ≤ 24 is 500 tuples. There is an unclustered B+ tree index on `B.author`, a clustered B+ tree index on `C.bid`, and all index pages are in memory. Also we assume unlimited memory to simplify things.

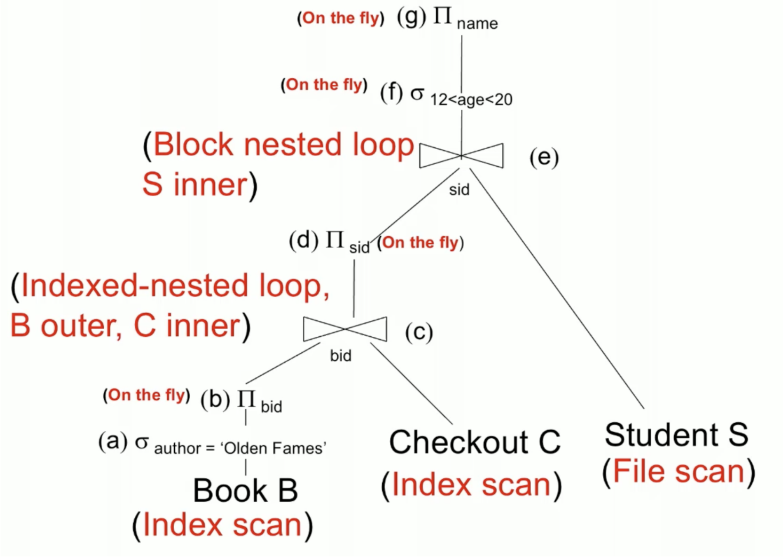

(On the fly) (g) Π_{name}
(On the fly) (f) $\sigma_{12<\text{age}<20}$
(Block nested loop S inner)
(e)
sid
(d) Π_{sid} (On the fly)
(Indexed-nested loop, B outer, C inner)
(c)
bid
(On the fly) (b) Π_{bid}
(a) $\sigma_{\text{author = 'Olden Fames'}}$
Book B
(Index scan)
Checkout C
(Index scan)
Student S
(File scan)

Figure 48: We have the following physical plan.

Okay, so let's do this. Note that since all index pages are in memory, we are storing the entire B+ tree in memory and don't need extra IOs to traverse it.

a) For the selection, the author is an unclustered B+ tree. There are 50,000 tuples and 500 distinct authors. Since we're querying 1 author, by uniformity we would need to access 100 book which may all be in their own disk page, so we need 100 IOs.[a] We end up with an output relation of 100 tuples in memory, so we also have a cardinality of 100.

$$\text{IO}(a) = \frac{T(B)}{500} = \frac{50,000}{500} = 100, \ \text{Card}(a) = 100 \tag{75}$$

b) For the projection on `bid`, we have already loaded in our relation in memory, so the IO cost is 0. We are still working with 100 pages, so our cardinality is still 100.

$$\text{IO} = 0, \ \text{Card}(b) = 100 \tag{76}$$

c) Now we do a join with an index-nested loop join on `bid`. Recall that we want to use the value of the outer table `R.bid` to probe the index on the inner table `C.bid`, which is clustered. We already have our outer table in memory, and we use the index `bid` to probe our inner table `C`. For each of the 50,000 book tuples in `B`, there are 300,000 checkout tuples in `C`, meaning that there are about $300,000/50,000 = 6$ checkout per book. For each of the 100 book tuples in (a), we expect to get 6 checkouts per book. There are $300,000/15,000 = 20$ checkout tuples per page, so counting for page boundaries we assume that 6 tuples will fit in at most 2 pages (or maybe 1). Therefore, we have

$$\text{IO}(c) = 100 \cdot 2 = 200, \ \text{Card}(c) = 100 \cdot 6 = 600 \tag{77}$$

d) This is done in memory so IO is 0. Note that we have a total of 600 checkout/book tuples. Since this is a projection, the cardinality also remains the same.

$$\text{IO}(d) = 0, \ \text{Card}(d) = 600 \tag{78}$$

e) Now we have a block nested loop join. Since (d) is already in memory (on the fly), all we have to do is load all of `S` into memory (unlimited), which means our IO cost is $B(S) = 1000$. We are joining with the student relation, and since there is 1 student per checkout, our output relation is still 600 tuples long.

$$\text{IO}(e) = 1000,\ \text{Card}(e) = 600 \tag{79}$$

f) Finally we select, and assuming that the ages are uniformly distributed, we expect $(20 - 12 - 1)/(24 - 7 + 1) = 7/18$ of the relations to remain after selection. IO is 0 since this is on the fly.

$$\text{IO}(f) = 0,\ \text{Card}(f) = 600 \cdot \frac{7}{18} \approx 234 \tag{80}$$

g) Finally, we project onto name. IO is 0 since on the fly. We are projecting on names and assuming we don't remove duplicates[b] our output relation is still the same length.

$$\text{IO}(g) = 0,\ \text{Card}(g) = 234 \tag{81}$$

The total cost is $1000 + 200 + 100 = 1300$ and the final cardinality is 234.

[a] If this was clustered, then each page can store 10 tuples, so we actually need 10 IOs.

[b] Is this really a realistic assumption?

6 XML

So far, we have talked about relational data, which is a type of **structured data** with a schema, attributes, tuples, etc along with their types and constraints. For example, just trying to add a new attribute to a relation may require you to add the attribute for each tuple if it cannot be null. Some **unstructured data** is just plaintext, which doesn't confirm to any schema, and is on the other extreme end. Some types of data in between is HTML, XML, or JSON, which we call **semi-structured**, which may contain sections, subsections, etc.

Definition 6.1 (XML)

The **extensible markup language (XML)** is a semi-structured data that is similar to HTML, where the data self-describes the structure.[a]

1. There are **tags** marked by a start (`<tag>`) and end (`</tag>`) tags.
2. An **element** is enclosed by a pair of start and end tags(`<t>...</t>`), and elements can be nested (`<t><r></r></t>`or empty(`<t></t>`.
3. Elements can also have **attributes** (`<book ISBN="...", price="80.00">`).[b] Attributes must be unique, i.e. there cannot be duplicate attributes in a tag.

Note that there are some conventions that we must follow to get a **well-formed** XML document. First, the tags should be closed appropriately like parantheses matching, and we should not have the characters < or > as a part of the element. Rather, we should use `&lt;` and `&gt;`.

[a] The names and nesting of its tags describe its structure.

[b] This is not the same attributes in relational databases.

Example 6.1

An example of XML data regarding a book is

```xml
<?xml version="1.0" encoding="UTF-8"?>
<book>
    <title>The Great Gatsby</title>
    <author>
        <firstName>F. Scott</firstName>
        <lastName>Fitzgerald</lastName>
    </author>
    <published>
        <year>1925</year>
        <publisher>Charles Scribners Sons</publisher>
    </published>
    <genre>Literary Fiction</genre>
    <price currency="USD">14.99</price>
    <inStock>true</inStock>
    <rating>4.5</rating>
</book>
```

This can be represented by a tree.

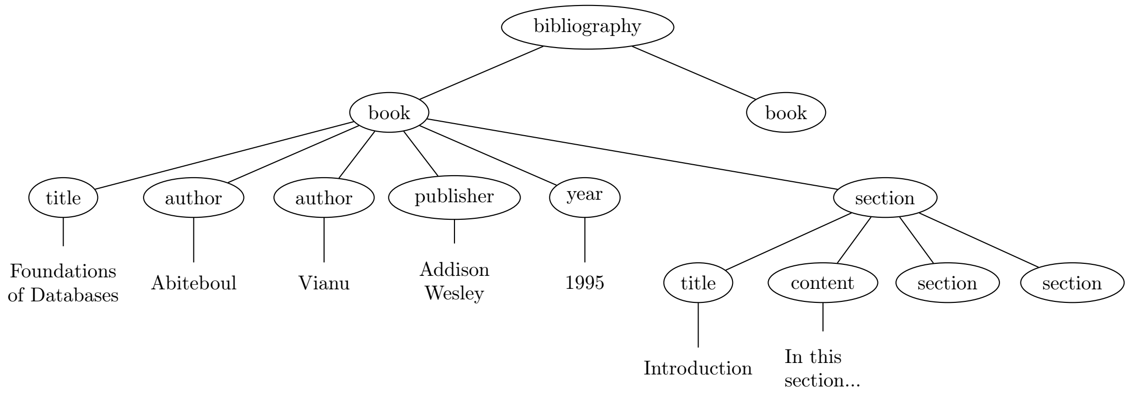


Figure 49: Tree representation of an XML file.

Let's do a quick comparison of relations and XML.

1. One advantage of XML is portability/exchangeability, since it self-describes itself, so all you really need is this text, unlike relations where you should also send the schema and other metadata. On the other hand, exchanging is problematic.
2. Second, its has flexibility to represent any information (e.g. structured, semi-structured, documents, etc.). Unlike a relation where each attribute of each tuple is atomic (meaning you can't place another relation in there), you can make a new XML tree.
3. Finally, since the data describes itself, you can change the schema easily. In contrast, the schema of a relational database is always fixed in advance and difficult to change.
4. Almost all major DBMS supports relations, and XML is often implemented as an add-on on top of relations.
5. Unlike the relational database where the order of the rows don't matter, the order does matter for XML.

Definition 6.2 (DTD)

Just because we don't need metadata does not mean that we cannot define one. This is done with the **document type definitions (DTD)**, which species the schema and constraints for XML just like relational databases. It has the following syntax, which should be written in the beginning of the XML file.

```
<!DOCTYPE root-element [
    <!ELEMENT element-name (content-specification)>
]>
```

Here are the common element specifications.
1. `EMPTY`: Element has no content
2. `ANY`: Element can contain any content
3. `(#PCDATA)`: Element contains parsed character data
4. `Child elements`: Listed in parentheses

with an example of a simple DTD being

```
<!DOCTYPE library [
    <!ELEMENT library (book+)>
    <!ELEMENT book (title, author, year)>
    <!ELEMENT title (#PCDATA)>
    <!ELEMENT author (#PCDATA)>
    <!ELEMENT year (#PCDATA)>
]>
```

Really, this is like a tree directory structure, so to query data, it makes sense to try and traverse the paths. There are three major query languages for XML, which are

1. **XPath**, which uses path expressions with conditions and are the building blocks of the rest of the standards.
2. **XQuery**, which is XPath plus a full-fledged SQL-like query language.
3. **XSLT**, or eXtensible Stylesheet Language Transformations and is the recommended style sheet language for XML.

6.1 XPath

Definition 6.3 (Basic XPath Constructs)

The syntax for building a XPath is as follows:

1. `/`: used as a separator between steps in a path (traversing down).
2. `name`: matches any child element with this tag name.
3. `*`: matches any child element
4. `@name`: matches the attribute with this name.
5. `@*`: matches any attribute
6. `//`: matches any descendent element of the current element itself.
7. `.` : matches current element.
8. `..` : matches parent element.[a]

Note that these names are all case sensitive, and another important fact is that NULL evaluates to NOT (so if there are no elements/attributes), then a condition on that element/attribute will evaluate to NOT.

[a] This is just like when we are navigating directories in a shell.

Example 6.2

Let's look at this example, which shows bibliographies for different books.

```
<?xml version="1.0" encoding="UTF-8"?>
<bibliography>
    <book ISBN="ISBN-10" price="70">
        <title>Foundations of Databases</title>
        <author>Abiteboul</author>
        <author>Hull</author>
        <author>Vianu</author>
        <publisher>Addison Wesley</publisher>
        <year>1995</year>
        <section>abc</section>
    </book>
    <book ISBN="ISBN-11" price="20">
        <title>DBSTS</title>
        <author>Ramakrishnan</author>
        <author>Gehrke</author>
        <publisher>Addison Wesley</publisher>
        <year>1999</year>
        <section>
          <title>bruh</title>
        </section>
    </book>
    <report>
      <author>Muchang</author>
      <addon>
        <author>Jon</author>
      </addon>
    </report>
</bibliography>
```

Let's look at some XPaths.

1. `/bibliography/book/author` gets all author element reachable via this path.

```
<author>Abiteboul</author>
<author>Hull</author>
<author>Vianu</author>
<author>Ramakrishnan</author>
<author>Gehrke</author>
```

2. `/bibliography/book/title` gets all book titles reachable via this path.

```
<title>Foundations of Databases</title>
<title>DBSTS</title>
```

3. `/bibliography/book/@ISBN` gets all book ISBN attributes (we need the to search the *attribute*, not the element)

```
ISBN="70"
ISBN="ISBN:11"
```

4. `//title` returns all title elements, anywhere in the document. Note that the title of the section is also added.

```
<title>Foundations of Databases</title>
<title>DBSTS</title>
<title>bruh</title>
```

5. `//section/title` returns all section titles, anywhere in the document.

```
<title>bruh</title>
```

6. `/bibliography/*/author` returns all authors in bibliography that is a grandchild of bibliography. Note that Muchang is retrieved but not Jon.

```
<author>Abiteboul</author>
<author>Hull</author>
<author>Vianu</author>
<author>Ramakrishnan</author>
<author>Gehrke</author>
<author>Muchang</author>
```

7. `/bibliography/book[@price<50]` returns all books that are children of bibliography with at-

tribute price less than 50.

```
<book ISBN="ISBN-11" price="20">
    <title>DBSTS</title>
    <author>Ramakrishnan</author>
    <author>Gehrke</author>
    <publisher>Addison Wesley</publisher>
    <year>1999</year>
    <section>
      <title>bruh</title>
    </section>
</book>
```

8. `/bibliography/book[author=’Abiteboul’]` returns all books that are children of bibliography that has an author tag children with element Abiteboul.
9. `/bibliography/book[author]` returns all books that are children of bibliography with some child author tag.
10. `/bibliography/book[40 <= @price and @price <= 50]` returns books with price attribute between 40 and 50.
11. `/bibliography/book[author=’Abiteboul’ or @price >=50]` returns books either authored by Abiteboul or price at least 50. Note that the first condition is on an element and the second is on an attribute.
12. `/bibliography/book[author=’Abiteboul’ or not (@price < 50)]` returns books authored by Abiteboul or price not below 50. Note that this is different from the query above since if a book does not have a price, then `price < 50` is null (which is NOT in XPath), so it returns the book.

Hopefully you get a good feel for how XPaths work. Here’s a trickier example.

Example 6.3 (Conditions on Any vs All Attributes/Elements)

Say you want to get all books with some price in the range $[20, 50]$. Then you would think of writing

```
/bibliography/book/[price>=20 and price <=50]
```

This may not work if we have a XML element of this form, with multiple prices.

```
<book>
  <title>newbooktitle</title>
  <price>10</price>
  <price>70</price>
</book>
```

since it compares is any element satisfies the conditions. Therefore, we can think of these conditions all as the `any` condition over child elements. If we want to use the `all` condition, we can write

```
/bibliography/book/[price[.>=20 and .<=50]]
```

Similarly, if we have the query `/bibliography/book[author=’A’ and author!=’A’]`, this is an any clause, so it will return all books with author A and another author not A.

Definition 6.4 (XPath Operators and Functions)

In conditions, we can use the following operations.

1. Arithmetic: `x + y`, `x - y`, `x * y`, `x div y`, `x mod y`
2. `contains(x, y)` returns true if string `x` contains string `y`
3. `count(node-set)` counts the number of nodes in `node-set`
4. `position()` returns the *context position* (i.e the position of the node amongst its siblings)
5. `last()` returns the *context size* (roughly the size of the node-set)
6. `name()` returns the tag name of the current element

Example 6.4

Some more queries is

1. `/bibliography/book[count(section)<10]` returns books with fewer than 10 sections.
2. `//*[contains(name(), 'Ab')]` returns all elements whose tag names contain `Ab` (or `section`?)
3. `/bibliography/book/section[position()=1]/title` returns the title of the first section in each book.
4. `/bibliography/book/section[position()=last()]/title` returns the title of the last section in each book.

6.2 XQuery

XQuery is a superset of the XPath, but it also encompasses FLWOR expressions, quantified expressions, aggregation, sorting, etc. An XQuery expression can even return a new result XML document. Let's put the XML example file again for convenience.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<bibliography>
    <book ISBN="ISBN-10" price="70">
        <title>Foundations of Databases</title>
        <author>Abiteboul</author>
        <author>Hull</author>
        <author>Vianu</author>
        <publisher>Addison Wesley</publisher>
        <year>1995</year>
        <section>abc</section>
    </book>
    <book ISBN="ISBN-11" price="20">
        <title>DBSTS</title>
        <author>Ramakrishnan</author>
        <author>Gehrke</author>
        <publisher>Addison Wesley</publisher>
        <year>1999</year>
        <section>
          <title>bruh</title>
        </section>
    </book>
</bibliography>
```

Example 6.5

For now, let's start with a simple XQuery based on XPath. To find all books, we can write the following, where the first `doc` refers to the specific document to query.

```
<result>{
  doc("bib.xml")/bibliography/book
}</result>
```

Note that like Python string interpolation, text outside of `{}` are copied to output verbatim, while those inside are evaluated and replaced by the result. We can use conditionals the same way.

```
<result>{
  doc("bib.xml")/bibliography/book[@price<50]
}</result>
```

which will return

```
<book ISBN="ISBN-11" price="20">
    <title>DBSTS</title>
    <author>Ramakrishnan</author>
    <author>Gehrke</author>
    <publisher>Addison Wesley</publisher>
    <year>1999</year>
    <section>
      <title>bruh</title>
    </section>
</book>
```

Definition 6.5 (FLWOR Expressions)

The fundamental building block of XQuery is the FLWOR expression, which stands for:

1. `FOR`: Iterates over sequences
2. `LET`: Binds variables to values
3. `WHERE`: Filters the tuples
4. `ORDER BY`: Sorts the results
5. `RETURN`: Constructs the result

Note that only the `RETURN` clause is mandatory; all others are optional.

Example 6.6 (FLWOR with Loops and Conditionals)

To retreive the titles of books published before 2000, together with their publisher, we can write either of the following. The logic is pretty self explanatory.

```
<result>{
  for $b in /bibliography/book
  let $p := $b/publisher
  where $b/year < 2000
  return
    <book>
    { $b/title }
    { $p }
    </book>
}</result>
```

```
<result>{
  for $b in /bibliography/book[year<2000]
  return
    <book>
    { $b/title }
    { $p/publisher }
    </book>
}</result>
.
.
```

which will return

```
<result>
```

```
  <book>
    <title>Foundations of Databases</title>
    <publisher>Addison Wesley</publisher>
  </book>
  <book>
    <title>DBSTS</title>
    <publisher>Addison Wesley</publisher>
  </book>
</result>
```

Note that `$p` may be assigned to a set of elements. It does not have to be one element.

Just like XPath, a conditional expression (with where) only checks the `any` condition.

Example 6.7 (Nested Loops and List Comprehension)

We can also use nested loops to solve the query above, but note the logic. If a book of price $<$ 2000 has 2 publishers, then the book may be returned 2 times. On the other hand, if a book has a price $<$ 2000 but has no publishers, then a null is really a false, so that book will not be returned.

```
<result>
{
  for $b in /bibliography/book,
  $p in $b/publisher
  where $b/year < 2000
  return
    <book>{ $b/title }{ $p }</book>
}
</result>
.
.
.
.
.
```

```
<result>
  <book>
    <title>Foundations of
     Databases</title>
    <publisher>Addison Wesley</publisher>
  </book>
  <book>
    <title>Foundations of
     Databases</title>
    <publisher>Springer</publisher>
  </book>
  <book>
    <title>DBSTS</title>
    <publisher>Addison Wesley</publisher>
  </book>
</result>
```

If we use a comprehension statement, we can let `$b` be all the books and try to parse them element by element where its year is $<$ 2000. This is also wrong since the where clause is an any expression, so if at least 2 book has year $<$ 2000, then it will return all the books. In fact, it doesn't even return it in the right format, since it returns all the book titles first and then all the publishers.

```
<result>
{
  let $b := /bibliography/book
  where $b/year < 2000
  return
    <book>
      { $b/title }
      { $b/publisher }
    </book>
}
</result>
```

```
<result>
  <book>
    <title>Found of Databases</title>
    <title>DBSTS</title>
    <publisher>Addison Wesley</publisher>
  </book>
</result>
.
.
.
.
```

Now that we went over FLWOR, let's talk about how we can do joins.

Example 6.8 (Explicit Join)

To find all pairs of books that have common authors, we can write

```
<result>
{
  for $b1 in //book
  for $b2 in //book
  where $b1/author = $b2/author
  and $b1/title > $b2/title
  return
    <pair>
      { $b1/title }
      { $b2/title }
    </pair>
}
</result>
```

Remember that since the where is an any condition, this works to our advantage now. We also use a string comparison of the authors and titles to remove duplicates. This gives us the result.

```
<result>
  <pair>
    <title>Foundations of Databases</title>
    <title>DBSTS</title>
  </pair>
</result>
```

We will stop here, but there are more features such as subqueries, existential (some) vs universal (all) queries, aggregation, conditional, etc.

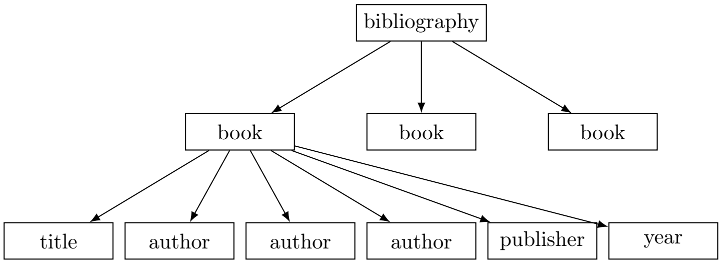
6.3 Converstion of XML to Relational Data

Relational to XML is trivial, but XML to relational isn't really since there can be different data types that are siblings (e.g. books vs articles, vs papers all under bibliography), there may be repeats of the same tag, and there are both child elements and attributes.

The most trivial thing to do is to store the entire XML in a column (called a CLOB, character large Object type), but this isn't useful since it first does not satisfy the condition that each element must be atomic in a relation. There are two alternatives.

1. **Schema-Oblivious Mapping** takes a well-formed XML and converts it to a generic relational schema. In here, there are more subtypes.
   (a) **Node/edge based mapping** for graphs.
   (b) **Interval based mapping** for trees.
   (c) **Path based mapping** for trees.
2. **Schema-Aware Mapping** takes a valid XML and converts it to a special relational schema based on DTD.

Let's go over the node/edge based mapping. This is quite intuitive since we can visualize a XML structure as a directed graph.



(a) XML document hierarchy as a tree structure

```
<bibliography>
<book ISBN="ISBN-10" price="80.00">
<title>Databases</title>
<author>Abiteboul</author>
<author>Hull</author>
<author>Vianu</author>
<publisher>Addison </publisher>
<year>1995</year>
</book>
</bibliography>
```

(b) XML document code representation

Figure 50: Two representations of an XML document: hierarchical tree structure (left) and code format (right)

We can convert the XML to the following 4 relations. We also put the bibliography example to make it easier to follow along.

1. **Element(eid, tag)**. All elements will have a certain id with their tag type. Note that elements can have text inside of them, but we will consider this a "child" of the element, stored in the `Text` relation.

Table 11: Element Table

eid	**tag**
e0	bibliography
e1	book
e2	title
e3	author
e4	author
e5	author
e6	publisher
e7	year

Table 12

2. **Attribute(eid, attrName, attrValue)**. All attributes will need to have their name and type, along with which eid that they are a part of. Note that there is only one functional dependency `(eid, attrName) -> attrValue`.

Table 13: Attribute Table

eid	**attrName**	**attrValue**
e1	ISBN	ISBN-10
e1	price	80

3. **ElementChild(eid, pos, child)**. Each element will have a children, with pos referring to the position of the children. Child references either `Element(eid)` or `Text(tid)`.

Table 14: ElementChild Table

eid	pos	child
e0	1	e1
e1	1	e2
e1	2	e3
e1	3	e4
e1	4	e5
e1	5	e6
e1	6	e7
e2	1	t0
e3	1	t1
e4	1	t2
e5	1	t3
e6	1	t4
e7	1	t5

4. **Text(tid, value)**. All text in an element, with an id value (which cannot be the same as any eid) and the actual text in the value.

Table 15: Text Table

tid	value
t0	Foundations of Databases
t1	Abiteboul
t2	Hull
t3	Vianu
t4	Addison Wesley
t5	1995

Note that we need to invent a lot of ids and need indices for efficiency.

Given this, we can write the equivalent SQL queries from these XPath queries.

1. `//title`

```sql
SELECT eid FROM Element WHERE tag='title';
```

2. `//section/title`

```sql
SELECT e2.eid
FROM Element e1, ElemtnChild c, Element e2
WHERE e1.tag = 'section'
AND e2.tag = 'title'
AND e1.eid = c.eid
AND c.child = e2.eid;
```

Therefore, path expression becomes joins. The number of joins is proportional to the length of the path expression.

7 JSON

NoSQL just stands for not SQL or not relational, like XML or **JSON**.[5] This relaxes some constraints and may be more flexible/efficient, with some popular data stores being MongoDB, CouchDB, Dynamo, etc. They are designed to scale simple OLTP (online transaction processing) style application loads and provide good horizontal scalability. An example of where this is used is in pretty much all blockchains. Every transaction and block is accessible in a JSON format on say `etherscan.io`.

Definition 7.1 (JSON)

A **JSON**, short for **JavaScript Object Notation**, data model is an object with the following properties:

1. At the top level, it is an array of *objects*.
2. Each object contains a set of key-value pairs of form `{key : value}`, where key (attribute) names should be unique within an object.
3. It supports most primitive types (numbers, strings, double, booleans are stored in quotes e.g. `"true"`), along with arrays `[...]`, and finally **objects** `{...}` (which defines a recursive structure).
4. The order is unimportant.
5. You can't comment in JSON files.

Example 7.1 (Example JSON)

Here is an example with users, groups, and members.

```
{
  "users": [
    {"uid": 1, "name": "Bart"},
    {"uid": 2, "name": "Lisa"}
  ],
  "groups": [
    {"gid": 101, "title": "Skateboarding Club"},
    {"gid": 102, "title": "Chess Club"}
  ],
  "members": [
    {"uid": 1, "gid": 101},
    {"uid": 1, "gid": 102},
    {"uid": 2, "gid": 102}
  ]
}
```

Definition 7.2 (MongoDB Database)

A database has collections of similarly structured documents, similar to tables of records as opposed to one big XML document that contains all data.

1. **Database** is a list of collections. (analogous to a database)
2. **Collection** is a list a documents (analogous to a table)
3. **Document** is a JSON object (analogous to a row/tuple)

MongoDB actually stores this data with **BSON** (Binary JSON), which is the binary encoding of it.

MongoDB provides a rich set of operations for querying and manipulating data.

[5] I think it's called this because it's literally how JS objects are stored and printed.

Definition 7.3 (find())

The `find()` operation is MongoDB's basic query mechanism. It returns a cursor to the matching documents.

1. Takes a query document that specifies the selection criteria
2. Can include projection to specify which fields to return
3. Supports comparison operators like `$eq`, `$gt`, `$lt`
4. Can query nested documents and arrays

Example 7.2 (Basic Query)

Find all users named "Bart":

```
db.users.find({"name": "Bart"})
```

Find users with uid greater than 1:

```
db.users.find({"uid": {$gt: 1}})
```

Definition 7.4 (sort())

The `sort()` method orders the documents in the result set.

1. Takes a document specifying the fields to sort by
2. Use 1 for ascending order, -1 for descending
3. Can sort by multiple fields
4. Applied after the query but before limiting results

Example 7.3 (Sorting)

Sort users by name in ascending order:

```
db.users.find().sort({"name": 1})
```

Definition 7.5 (Aggregation Pipeline)

An aggregation pipeline consists of stages that transform sequences of documents. Each stage performs an operation on the input documents and passes the results to the next stage. Despite its name, it handles more than just aggregation operations.

MongoDB supports several types of pipeline stages:

1. **Selection and Filtering** (`$match`)
   - Filters documents based on specified conditions
   - Similar to `find()` but within the pipeline context
2. **Projection** (`$project`)
   - Reshapes documents by including, excluding, or transforming fields
   - Can create computed fields

3. **Sorting** (`$sort`)
   - Orders documents based on specified fields
   - Equivalent to the `sort()` method
4. **Grouping** (`$group`)
   - Groups documents by a specified expression
   - Supports aggregation operators:
     - `$sum`: Calculates numeric totals
     - `$push`: Accumulates values into an array
5. **Document Transformation**
   - `$project`/`$addFields`: Adds computed fields
   - `$unwind`: Deconstructs arrays into individual documents
   - `$replaceRoot`: Promotes an embedded document to the top level
   - Array operators:
     - `$map`: Applies an expression to each array element
     - `$filter`: Selects array elements matching a condition
6. **Joining** (`$lookup`)
   - Performs left outer joins with other collections

Example 7.4 (Basic Pipeline)

A pipeline that finds users in the "Chess Club" and sorts them by name:

```
db.members.aggregate([
    {$lookup: {
        from: "users",
        localField: "uid",
        foreignField: "uid",
        as: "user"
    }},
    {$match: {"gid": 102}},
    {$sort: {"user.name": 1}}
])
```

Example 7.5 (Complex Pipeline)

A pipeline using multiple stages to group and transform data:

```
db.members.aggregate([
    {$group: {
        _id: "$gid",
        members: {$push: "$uid"},
        count: {$sum: 1}
    }},
    {$lookup: {
        from: "groups",
        localField: "_id",
        foreignField: "gid",
        as: "group_info"
    }},
    {$project: {
        _id: 0,
        group: {$arrayElemAt: ["$group_info.title", 0]},
        member_count: "$count",
        members: 1
    }}
])
```

The pipeline stages must be carefully ordered as each stage's output becomes the input for the next stage. For optimal performance:

1. Use `$match` early to reduce the number of documents
2. Place `$project` and `$unwind` before `$group` when possible
3. Consider memory limitations when using `$sort`

Definition 7.6 ($lookup)

The `$lookup` operation performs a left outer join to another collection.

1. Must be used within an aggregation pipeline
2. Specifies foreign collection to join with
3. Defines local and foreign fields to join on
4. Results stored in an array field

Example 7.6 (Join Operation)

Join members with users:

```
db.members.aggregate([
  {$lookup: {
    from: "users",
    localField: "uid",
    foreignField: "uid",
    as: "user_info"
  }}
])
```

Definition 7.7 (Aggregation Operators)

Special operators used within aggregation pipelines for computations.

1. `$sum`: Calculates sum of numeric values
2. `$push`: Adds value to an array
3. `$avg`: Calculates average
4. `$first`/`$last`: Returns first/last value in group

Example 7.7 (Aggregation Operators)

Calculate total members and collect group IDs:

```
db.members.aggregate([
  {$group: {
    _id: "$uid",
    total: {$sum: 1},
    groups: {$push: "$gid"}
  }}
])
```

8 Transactions

So far, we've had one query/update on one machine, but this is not the case in reality. In modern systems, you have users interacting with a database all the time and databases may be prone to failure. These two situations requires us to develop a safer way of interacting with the database.

Example 8.1 (Simultaneous Interaction)

In an airline, say that we have two parties A and B trying to book the same seat. A looks at the website, which queries the tuple representing the seat. B does the same. A and B then both book it at the same time, sending an update request to the database, causing an overbooking.
In a bank, say that we have some events that must be recorded in a database. Say that $A = 0$ and $B = 100$.

1. T1. B sends \$100 to A. So $A \mapsto A + 100, B \mapsto B - 100$.
2. T2. The bank sends all users an interest of 6%. $A \mapsto 1.06A, B \mapsto 1.06B$.

This is sensitive to order and perhaps at the end we expect $A = 100$. Consider the following cases.

1. $A \mapsto A + 100, B \mapsto B - 100, A \mapsto 1.06A, B \mapsto 1.06B$. This is fine.
2. $A \mapsto A + 100, A \mapsto 1.06A, B \mapsto 1.06B, B \mapsto B - 100$. This is not fine since both A and B got interest and the bank lost \$6.

Example 8.2 (Database Crash)

If $A \mapsto A + 100$ happened first and the bank crashed, then the system would find that A just gained \$100! This is clearly not good, so we must undo what we have done so far.

These two examples hint at some nice properties that we want in our database.

1. *Serializability.* The first example shows that we do not like parallelism and rather we want things to run *serially* in a way such that T1 and T2 occur separately, like a mutex lock. In practice, there are so many requests that parallelism is a must, so we must find ways to not run serially, but to modify our parallel computing to make it serializable, as if it is running serially.

2. *Atomicity.* The second example shows that we want these operations to be *atomic*, i.e. it is fully done or not done at all. That is, we do not want to update the disk (which preserves its state upon powering off) until T1 is completely finished. A simple solution is to do all the operations in memory (which is wiped upon crashing anyways) and then *commit* (announce that it is done) these changes. [6]

Great, so let's review some actions that our DBMS should support. To read/write some tuple or relation X, the disk block containing X must first be brought into memory. X is read/written in memory. If it is written, it is called a **dirty block**. Finally, this dirty block must be flushed to disk.

Definition 8.1 (Actions)

Some examples of actions that can occur in a schedule is:

1. `READ`: read in a memory block from disk
2. `WRITE`: write/flush out a dirty memory block
3. `COMMIT`
4. `ABORT`

Note that committing is not the same as writing!

[6] Committing is not the same as disk writing, actually. It is not necessarily the case that one follows the other, so you can commit before writing and write before committing. There can also be some completed transactions with updated data still in memory and therefore lost in crash.

Definition 8.2 (Transaction)

The solution to both serializability and atomicity is to use **transactions**, which is a collection of one or more operations, called **actions**, on the database that must be executed atomically as a whole. At the end of every transaction, the DBMS always puts either a commit (indicating successful completion) or abort signal.

Definition 8.3 (Schedule)

Now that we've cleared up on transactions, a schedule is simply a sequence of actions gotten from interweaving the transactions on the DBMS. It must obviously be consistent, and two actions from the same transaction T must appear in the schedule in the same order that they appear in T. Some terminology to compare schedules:

1. **Serial Schedule**. No interweaving of transactions.
2. **Equivalent Schedule**. Two schedules that have the same effect on the database state after completion.
3. **Serializable Schedule**. A schedule that is equivalent to a serial schedule.

In a schedule, the transactions are usually written in $T, S, \ldots$ and they act on **elements** (data, tuples or relations, on disk or memory).

Example 8.3 (Bank Schedule)

For the bank example above, we can construct a serial schedule S_1.

```
T1: R(A) W(A) R(B) W(B) C(T1)
T2:                           R(A) W(A) R(B) W(B) C(T2)

T:  R1(A) W1(A) R1(B) W1(B) C(T1) R2(A) W2(A) R2(B) W2(B) C(T2)
```

and a serializable schedule S_2.

```
T1: R(A) W(A)           R(B) W(B) C(T1)
T2:           R(A) W(A)                 R(B) W(B) C(T2)
T:  R1(A) W1(A) R2(B) W2(B) R1(B) W1(B) C(T1) R2(B) W2(B) C(T2)
```

We can verify that $S_1 \equiv S_2$ since in the second transaction, we always read in A after it is written by the first transaction. However, the following is not a serializable schedule since T2 reads the unmodified A block from disk before flushed by T1.

```
T1: R(A)                           W(A) R(B) W(B) C(T1)
T2:      R(A) W(A) R(B) W(B) C(T2)
```

These properties are a subset of the general ones described here. In the next sections, we will go over implementing features that ensure each of these properties.

Definition 8.4 (ACID)

These properties are a subset of the **ACID** properties.

1. *Atomicity.* A user can think of a transaction as always executing all actions in one stop or not executing any at all. In an abort, this can be undone by storing recovery logs and going through the history to undo.

2. *Consistency.* Each transaction, when run by itself with no concurrent execution of other actions, must preserve the consistency of the database.[a]
3. *Isolation.* A user should be able to understand a transaction without considering the effect of any other concurrently running transaction. This allows us to have *concurrency control*, which handles multiple transactions by interweaving them in a serializable way (like the bank example we saw above), and sometimes they may ensure isolation with locks.
4. *Durability.* Once the DBMS informs the user that a transaction is completed, its effect should persist.[b]

It turns out that making one stricter usually leads to a decrease in performance in the other, so we should maintain some balance. C/I are usually improved in conjunction with concurrency controls, and A/D are improved using recovery.

[a] e.g. if you transfer money from one account to another, the total amount in circulation still remains the same. The constraints must also be maintained.

[b] Even if the system crashes before writing to disk, we can recover it by using the history logs and redoing them.

8.1 Consistency: Serizability and Precedence Graph

Okay, so we've seen in the bank schedule example that serial schedules are trivial to construct and ideally we want to construct serializable schedules. How do we detect them? Well it turns out checking for serializability is NP-complete, so we must resort to something called *conflict-serializability*, which is a stronger condition but can be solved in polynomial time. Conflict-serializability assures the absence of *conflicts*, which are certain properties of schedules that result in a corrupt schedule.

Definition 8.5 (Conflict)

When we interweave actions, **conflicts** cause schedules to be not equivalent to the serial schedule. There are 3 types of conflicts.

1. Write-Read (WR): T_1 writes A and then T_2 reads A before T_1 commits. This causes reading uncommitted (dirty) data.
2. Read-Write (RW): T_1 reads A and then T_2 writes A before T_1 commits. This causes unrepeatable reads.
3. Write-Write (WW): T_1 writes A and then T_2 writes A before T_1 commits. This overwrites uncommitted data or lost updates.
4.

Note that RR (read read) has no conflicts since there are no writes.

We can make a *precedence graph*.

Definition 8.6 (Precedence Graph)

A **precedence graph** contains

1. a node for each transaction
2. a directed edge from T_i to T_j if an operation of T_i precedes and conflicts with an operation of T_j in the schedule.

Example 8.4 (Precedence Graph)

Consider the following schedule between 2 transactions.

```
R1(A) R2(A) W1(A) C1 C2
```

Then, our precedence graph $G(V, E)$ is defined $V = \{T_1, T_2\}$ and $E = \{(T_2, T1)\}$ since there is a RW conflict. Now consider the following.

```
R1(A) R2(A) W1(A) W2(A) C1 C2
```

Its precedence graph has both edges (T_1, T_2) due to `R1(A)...W2(A)` and (T_2, T_1) due to `R2(A) W1(A)`.

Theorem 8.1 (Conflict-Serializability)

A schedule is **conflict-serializable** (which implies serizability) if its precedence graph is acyclic.

Example 8.5 (Not Serializable)

The following schedule is not conflict serializable since $T_1 \mapsto T_2$ from the transactions on A and $T_2 \mapsto T_1$ from the transactions on B.

```
T1: R(A) W(A)                     R(B) W(B)
T2:           R(A) W(A) R(B) W(B)
```

We established a method to check if a certain schedule S_1 is conflict-serializable. Given two serial schedules S_1, S_2, we can check if they are equivalent serial schedules as well.

Theorem 8.2 (Equivalent Serial Schedules)

Given 2 serial schedules S_1, S_2, let G_1, G_2 be their precedence graphs. If G_1 and G_2 have a common existing topological ordering, then $S_1 \equiv S_2$.

Theorem 8.3 (Swapping Non-Conflicting Actions Doesn't Change Precedence Graph)

You can also swap adjacent non-conflicting actions to reach an equivalent serial schedule.

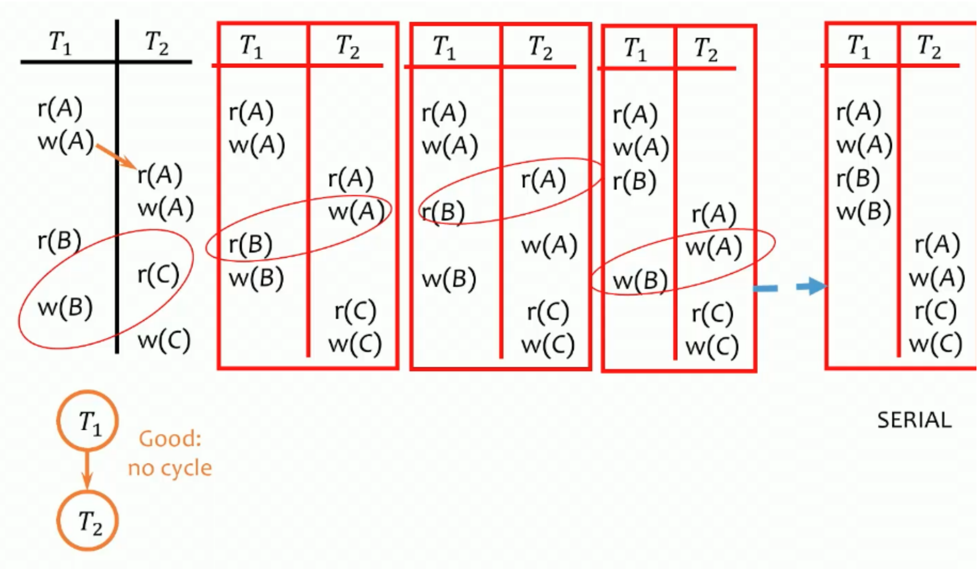


Figure 51: You can first swap W(B) and R(C) since they act on different elements. You keep swapping to push the actions of T1 up as far as possible, and if you can get a serial schedule, you are good. Swapping non-conflicting actions does not affect the precedence graph of the schedule(?)

8.2 Isolation and Concurrency Control

We've seen that ideally, we would like to have a schedule that is conflict-serializable. The avoiding of conflicts is just one problem that we must avoid when implementing isolation protocols, and in here we will focus on implementing a protocol that enforces three properties:

1. conflict-serializability
2. avoiding cascading rollbacks
3. recoverability

While looking at precedence graphs allow us to compare already existing schedules, it does not give us instructions to *create* one. We can do this using locks.

8.2.1 Locks

Let's introduce locks in the basic sense.

Definition 8.7 (Locks)

The rules for **locks** are as follows:

1. If a transaction wants to read an object, it must request a **shared lock** (S mode) on that object.
2. If a transaction wants to read/modify an object, it must first request an **exclusive lock** (X mode) on that object.
3. You can allow one exclusive lock, or multiple shared locks.

Example 8.6 (Basic Locking)

An implementation of basic locking is shown below.

```
T1: X(A) R(A) W(A) U(A)                                        X(B) R(B) W(B) U(B)
```

```
T2:                     X(A) R(A) W(A) U(A) X(B) R(B) W(B) U(B)
```

However, notice that basic locking does not really enforce conflict-serializability. Naive locks still allow for this non-serializable schedule to occur. Therefore, we can do two-phase locking.

Definition 8.8 (Two-Phase Locking (2PL))

The basic rule is that *for each individual transaction*, all lock requests precede all unlock requests. Therefore, every transaction is divided into a phase of obtaining locks and releasing locks.

Example 8.7 (2PL)

Look at this previous schedule.

```
T1: R(A) W(A)                     R(B) W(B)
T2:           R(A) W(A) R(B) W(B)
```

1. T1 will first requests a X lock on A.
2. Then T2 wants to write to A so it must have the lock. In order to do this T1 must unlock A, but as soon as it unlocks A, it cannot request for any more locks. What can it do? It has no choice but to request for a lock on B *now*, and then unlock A to give to T2.
3. T2 unlocks A, reads/writes it, and then wants to unlock A, but again it must request B before unlocking A. However, T1 already holds the lock on B and cannot unlock it since it must write B later.

```
T1: X(A) R(A) W(A) X(B) U(A)
T2:                          X(A) R(A) W(A) ... not allowed
```

This block really comes from the fact that this schedule is not serializable. Consider a serializable schedule instead.

```
T1: R(A) W(A)           R(B) W(B)
T2:           R(A) W(A)           R(B) W(B)
```

Then we can construct the 2P locks as such.

```
T1: X(A) R(A) W(A) X(B) U(A)                R(B) W(B) U(B)
T2:                          X(A) R(A) W(A)                X(B) U(A) R(B) W(B) U(B)
```

Fundamentally, what we are trying to do by having all locks come first is to push all actions on A (and their associated locks) as far up as possible so we can finish them first. By pushing them up, we can isolate R and not have two-way conflicts between T1 and T2.

2PL is great, but we still have to look at the remaining two desired properties. Let's show why it doesn't satisfy them.

Example 8.8 (Cascading Rollback and Irrecoverability in 2PL)

Consider the following problems with 2PL.

1. Say that T1 is reading A and adds 5 to it to write. Then T2 will read $A = 10$ and double it for write. If T1 decides to abort before writing, it should look like T1 had never happened, so T2

should have been reading $A = 5$. This is problematic, and the only way to deal with this is to abort T2 as well. If there are other transactions, this is called a **cascading rollback**.

```
T1: R(A=5) W(A=10)         Abort
T2:                R(A=10)       W(A=20) Must abort T2
```

2. A schedule may not be recoverable sometimes. We can see that T1 aborts, but by the time it aborts, it's too late: T2 had already committed changes using uncommitted data from T1, which must now be undone.

```
T1: R(A=5) W(A=10)                        Abort
T2:                R(A=10) W(A=20) Commit
```

Therefore, we would like to

1. *Avoid cascading rollbacks.* All transactions should only read data by committed transactions.
2. *Recoverable.* A transaction commits only after all transactions it had read from commits.

By enforcing even stricter conditions, we can get this as well.

Definition 8.9 (Strict 2PL)

Strict 2PL only releases locks at commit/abort time. A writer will block all other readers until the writer commits or aborts.

Example 8.9 (2PL vs Strict 2PL)

Consider the following schedule.

```
T1: X(A) W(A) U(A)                C
T2:                X(A) W(A) U(A)   C
```

1. 2PL allows this schedule since for each transaction, all unlocks come after all locks.
2. Strict 2PL does not allow this schedule since it asserts that you must commit before unlocking (and having other transactions use the lock).

With strict 2PL, you must use this schedule, which requires the commits to be before the transaction.

```
T1: X(A) W(A) C U(A)
T2:                  X(A) W(A) C U(A)
```

It seems that we are regressing more and more back to the serial schedule, and yes this is true. By enforcing stricter conditions, we are forced to isolate more and more until it is serial. However, this is used in many commercial DBMS because reliability is often more important than efficiency.[7]

8.2.2 Snapshots

Oracle uses snapshot isolation. Covered in 516. It uses a private snapshot or a local copy. If there are no conflicts, it makes global changes and aborts otherwise. It is more efficient than locks, but may lead to aborts.

[7] Oracle is a notable exception.

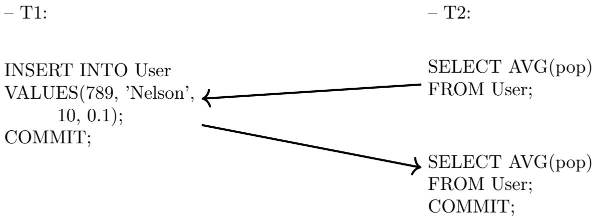
8.2.3 Timestamp-Based Concurrency Control

8.2.4 Conflicts and Isolation Levels

We can actually enforce isolation levels in SQL directly. In SQL a transaction is automatically started when a user executes a SQL statement, and subsequent statements in the same session are executed as part of this transaction. Statements see changes made by earlier ones in the same transaction and statements in other concurrently running transactions do not. `COMMIT` and `ROLLBACK` commands are self explanatory.

SQL also supports isolation levels, which increases performance by eliminating overhead and allowing higher levels of concurrency, albeit at the cost of getting inconsistent answers if you don't know what you're doing. We start off with the most restrictive and go down.

1. `SERIALIZABLE` is the strongest isolation level, with complete isolation.
2. `REPEATABLE READ` allows repeated reads, and you are safe with updates, but you aren't safe with new insertions, which are called *phantoms*.

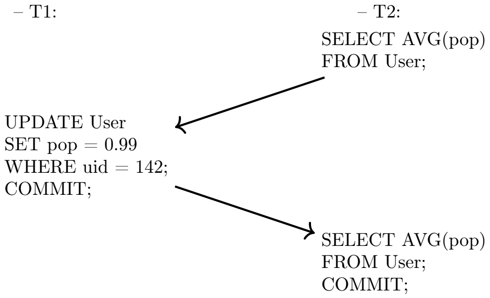


```
-- T1:                          -- T2:

                                SELECT AVG(pop)
INSERT INTO User                FROM User;
VALUES(789, 'Nelson',
       10, 0.1);
COMMIT;
                                SELECT AVG(pop)
                                FROM User;
                                COMMIT;
```

Figure 52: You still have a different average due to the insertion of a new element.

3. `READ COMMITTED` does not allow dirty reads, but non-repeatable reads are allowed (RW conflicts), which means that reading the same data twice can produce different results and is allowed.



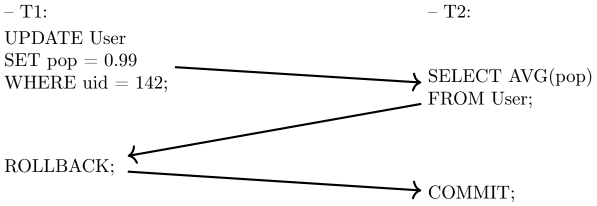
```
-- T1:                  -- T2:
                        SELECT AVG(pop)
                        FROM User;

UPDATE User
SET pop = 0.99
WHERE uid = 142;
COMMIT;

                        SELECT AVG(pop)
                        FROM User;
                        COMMIT;
```

Figure 53: In T2, we first compute an average of say 0.6. Then in T1 we update the popularity and commit. Then in T2, we take the average again, getting 0.8, and finally commit. T2 ends up reading two different states of the database.

4. `READ UNCOMMITTED` allows you to read uncommitted/dirty data (WR conflict).



```
– T1:
UPDATE User
SET pop = 0.99
WHERE uid = 142;

ROLLBACK;
```

```
– T2:
SELECT AVG(pop)
FROM User;

COMMIT;
```

Figure 54: In T1, you are setting the popularity to 0.99. In T2, you read this uncommitted data to compute the average. However, T1 aborts this write, and in T2 you are left with the wrong average.

Here is a summary of the isolation levels.[8]

Isolation level/anomaly	**Dirty reads**	**Non-repeatable reads**	**Phantoms**
READ UNCOMMITTED	Possible	Possible	Possible
READ COMMITTED	Impossible	Possible	Possible
REPEATABLE READ	Impossible	Impossible	Possible
SERIALIZABLE	Impossible	Impossible	Impossible

Figure 55: SQL Transaction Isolation Levels and Possible Anomalies

8.3 Atomicity and Durability

Note that whenever we read from or write to disk, we are incurring an IO cost and therefore a delay. Since committing and writing to disk are not the same thing, there are two risks:

1. A system may crash in the middle of transaction T before committing, keeping only partial effects of T. This violates atomicity, and we should know how to *undo* T.
2. A system may crash right after a transaction T commits, and not all effects of T were written to disk. This violates durability, and we should know how to *complete* T.

Example 8.10 (No Steal and Force)

A naive approach to these two risks are as such.

1. *No steal.* Writes of a transaction can only be flushed to disk at commit time. With steal, if the system crashes before T commits but after some writes of T have been flushed to disk, there is no way to undo these writes.
2. *Force.* When a transaction commits, all writes of this transaction must be reflected on disk. Without force, if a system crashes right after T commits, the effects of T will be lost.

8.4 Logging

Just as we enforced isolation and concurrency control with locking, we can kill two birds with one stone here with *logging*.

Definition 8.10 (Log)

A **log** is a sequence of **records** that records all changes made to the database.[a] There are 3 types of logs.

1. `Start`: When a transaction starts, we log

$$\langle T_i, \texttt{start} \rangle \tag{82}$$

[8] Postgres defaults to read committed.

2. `Write`: When transaction T takes element A and changes value 5 to 10, we log
$$(T, A, 5, 10) \tag{83}$$
3. `Commit`, `Abort`: When a transaction T is committed or aborted, we log
$$\langle T_i, \texttt{commit}\rangle \text{ or } \langle T_i, \texttt{abort}\rangle \tag{84}$$

Logs are stored in disk (stable storage) and used in recovery. It satisfies **write-ahead logging (WAL)**, which states that before A is modified in disk, the log record pertaining to A must be flushed.

[a] This is stored in a special data structure called *aries*, developed at IBM.

At first glance, logging may sound like a bad idea. Since every time we do a transaction, we don't do 1 IO but 2, so this may hurt our performance. However, log writing is really sequential since we're appending to the end of a log only, so we can use dedicated disks with fast append operations to minimize this performance impact.

Theorem 8.4 (No Force and Seal of Logging)

Counterintuitively, logging applies no force and steal (as opposed to no steal and force in our naive approach).

1. *Steal*. Modified memory blocks can be flushed to disk anytime. This is fine since undo information is logged due to WAL.
2. *No Force*. A transaction can commit even if its modified memory blocks have not been written to disk. This is fine since redo information is logged due to WAL.

Example 8.11 (Bank Transfer)

Say that we have two elements/tuples $A = 800, B = 400$ on disk. The schedule is just one transaction that will look something like this. Let D, L, M refer to our main disk, log disk, and memory.

1. We log that T has started.

```
Disk    Memory
A=800
B=400
```

```
Log
[T, start]
.
```

2. T loads $A = 800$ from disk to memory.

```
Disk    Memory
A=800   A=800
B=400
```

```
Log
[T, start]
.
```

3. T writes $A = 700$ in memory and logs the write at the same time. This is fine since WAL forces logging before *flushing* to disk, not writes on memory.

```
Disk    Memory
A=800   A=700
B=400
```

```
Log
[T, start]
[T, A, 800, 700]
```

4. We load $B = 400$ into memory.

```
Disk    Memory
A=800   A=700
B=400   B=400
```

```
Log
[T, start]
[T, A, 800, 700]
```

5. T writes $B = 500$ in memory and logs the write.

```
Disk    Memory
A=800   A=700
B=400   B=500
.
```

```
Log
[T, start]
[T, A, 800, 700]
[T, b, 400, 500]
```

6. We flush the value of A to disk. Note that by steal, we can flush before committing T.

```
Disk    Memory
A=700   A=700
B=400   B=500
.
```

```
Log
[T, start]
[T, A, 800, 700]
[T, b, 400, 500]
```

7. We commit T, which goes to the log. This marks the end of the transaction.

```
Disk    Memory
A=700   A=700
B=400   B=500
.
```

```
Log
[T, start]
[T, A, 800, 700]
[T, b, 400, 500]
[T, commit]
```

8. We flush the value of B to disk. Note that by no force, we can flush after committing T. Note that this does not log.

```
Disk    Memory
A=700   A=700
B=500   B=500
.
```

```
Log
[T, start]
[T, A, 800, 700]
[T, b, 400, 500]
[T, commit]
```

With this flexibility to flush anytime, we as the programmers really just care about the correctness about the schedule and don't care about the order.

8.4.1 Checkpointing

We've seen the advantages of log files, but they can get quite long very fast. If a system crashes and we must recover our operations, it may not be ideal to start the recovery from the beginning of a schedule. Rather, we want to use *checkpointing*. Let's go through a naive approach.

Definition 8.11 (Naive Checkpointing)

To checkpoint,

1. we stop accepting new transactions
2. finish all active transactions
3. take a database dump

To recover, we can start from the last checkpoint.

We can already see a few problems with this approach, especially the performance impact on not accepting new transactions. Therefore, we use a more sophisticated approach.

Definition 8.12 (Fuzzy Checkpointing)

To place a fuzzy checkpoint, we do the following.

1. We determine S, the set (ids of) currently active transactions.

2. We log ⟨`START CKPT S`⟩
3. We flush all dirty blocks up to the checkpoint to disk (however, actions within the checkpoint may or may not get flushed within the checkpoint).
4. Transactions normally proceed and new transactions can start during checkpointing.
5. We log ⟨`END CKPT START-CKPT_location`⟩, where the start checkpoint location allows you to easily access the location of the start (otherwise we can read the log backwards to find it).

Now once a bad event like a crash happens, there are three recovery steps.

1. *Analysis.* Go backward from crash point.
2. *Repeating History.* Forward takes care of REDO for committed transactions.
3. *UNDO.* Backward takes care of UNDO of uncommitted transactions and removes their effects.

Example 8.12 (Crash After T2)

Consider the logs below. The left log shows the full log, and the right log shows a log up until a system crash right before committing transaction T3.

```
1  <START T1>
2  <T1, A, 4, 5>
3  <START T2>
4  <COMMIT T1>
5  <T2, B, 9, 10>
6  <START CKPT(T2)>
7  <T2, C, 14, 15>
8  <START T3>
9  <T3, D, 19, 20>
10 <END CKPT>
11 <COMMIT T2>
12 <COMMIT T3>
```

```
1  <START T1>
2  <T1, A, 4, 5>
3  <START T2>
4  <COMMIT T1>
5  <T2, B, 9, 10>
6  <START CKPT(T2)>
7  <T2, C, 14, 15>
8  <START T3>
9  <T3, D, 19, 20>
10 <END CKPT>
11 <COMMIT T2>
12 -- CRASH
```

In the full log, we see that

1. At the time of checkpoint, T2 is active but T1 is already committed, so the current set of transactions is just T2.
2. All logs before `START CKPT` goes to disk.
3. During the checkpoint, T2 writes to C.
4. We can also see that during the checkpoint, a new transaction T3 starts.
5. Then we can commit T2 and T3.

In the log with the crash, we go through the recovery steps.

1. We look at line 11 at the end of the log where T2 commits. We go back until we reach an `END CKPT` and use it to go back to `START CKPT` at line 6. If there is no `CKPT`, then go back to the beginning of the log.
2. Now for the redo step.
   (a) After analysis, we first construct a set U of uncommitted transaction to be used in the UNDO step later. Initially, $U = S = \{T_2\}$ since T_1 has already committed and writes are already on disk.
   (b) We scan forward from `START CKPT(T2)` and repeat every action (redo) from here until the end of the log. If we see a log record of form `<T, A, old, new>`, we can simply write `(A, new)`, flush to disk, and ignore the old value, as it may or may not be preserved (depending on whether we flushed before crash or not).
   (c) We see that T3 starts at line 8, so we add T_3 to U. If a transaction T committed or aborted, we remove T from U.
   (d) We see after the `CKPT` that T2 is committed. We remove T_2 from U.
3. Finally, we do the undo step since some transactions are still uncommitted. We run backwards from the end of log, to the earliest `<START T>` of the uncommitted transactions stored in set U (which may be before or after the `START CKPT`).

(a) We have $U = \{T_3\}$, and so we go backwards an undo all the writes. That is, for each log record `<T, A, old, new>`, where $T \in U$, we can simply write `(A, old)` and log this operation.
(b) We do this until we hit the log `START T3` at line 8.
(c) We finally log `<abort T>` when all effects of T have been undone.

Example 8.13 (Crash After T2 and T3)

Say that our crash happens after committing T3.

```
1  <START T1>
2  <T1, A, 4, 5>
3  <START T2>
4  <COMMIT T1>
5  <T2, B, 9, 10>
6  <START CKPT(T2)>
7  <T2, C, 14, 15>
8  <START T3>
9  <T3, D, 19, 20>
10 <END CKPT>
11 <COMMIT T2>
12 <COMMIT T3>
13 -- CRASH
```

In this case, $U = \{T_2\}$ after analysis, and after redo, $U = \{\}$, and so there is no undo phase.

Example 8.14 (Crash After T3)

Consider when we commit T3 and then it crashes before committing T2.

```
1  <START T1>
2  <T1, A, 4, 5>
3  <START T2>
4  <COMMIT T1>
5  <T2, B, 9, 10>
6  <START CKPT(T2)>
7  <T2, C, 14, 15>
8  <START T3>
9  <T3, D, 19, 20>
10 <END CKPT>
11 <COMMIT T3>
12 -- CRASH
```

T1 has committed and writes are already on disk. After analysis, $U = S = \{T_2\}$. When we do redo all actions, we again take line 7 and write `T2(C=15)`, then we add T_3 to U, then we write `T3(D=20)`. Then we end checkpoint and commit T3, removing T_3 from U. We have $U = \{T_2\}$ after redo, so in our undo phase, we go backwards.

1. In line 7, we write `T2(C=14)` (which was already flushed to disk by redo step) and flush to disk.
2. Beyond the checkpoint start, we write `T2(B=9)` (which was already flushed to disk since it's before the start CKPT) and flush to disk.
3. We finally write `<abort T2>` at the end of the checkpoint

We are done, and the effects of T2 have vanished.

Example 8.15 (Crash Before T2 and T3)

Now we consider when we have a crash before both commits.

```
<START T1>
<T1, A, 4, 5>
<START T2>
<COMMIT T1>
<T2, B, 9, 10>
<START CKPT(T2)>
<T2, C, 14, 15>
<START T3>
<T3, D, 19, 20>
<END CKPT>
-- CRASH
```

The analysis is the same with $U = \{T_2\}$. Then we redo and flush changes to disk up until the end and find $U = \{T_2, T_3\}$. Finally, we have to undo these uncommitted transactions, where we go up until `<START T2>`, and rewrite the old values and flush them back the disk.

9 Big Data

As powerful relational models are, the sheer growth of data in modern times requires the use of hundreds or thousands of database servers working together to modify or query data. This field is known as *big data*. Unlike transactions, which deal with multiple queries on one machine, big data deals with one query on multiple machines.

9.1 Parallel Databases

9.2 Map-Reduce