Contents

1. File Types

1.1 CSV Files

- CSV stands for comma separated values
- It is a flat file format (no nesting) consisting of records separated by newlines, with each field separated by commas
  - ↳ Fields can be enclosed in double quotes, but need not be unless a comma, double quote, or newline occurs
  - ↳ Double quotes are escaped with another double quote
  - ↳ Nested structure must be represented using text or strings, requiring additional parsing
- The first line of the file may be a header line, providing names for each field

1.2 JSON Files

- JSON stands for JavaScript Object Notation
  - ↳ While originally derived from JavaScript, it is now a language-independent data format
- Data is contained in a semi-structured hierarchical structure under one root JSON object
  - ↳ Whitespace outside strings is ignored
- The types in JSON are:
  - `"string"`, a sequence of zero or more characters enclosed in double quotes
    - ↳ Characters are escaped with `\`, such as `"`, `\`, and optionally `/`
    - ↳ Escape sequences include backspace `\b`, formfeed `\f`, newline `\n`, carriage return `\r`, tab `\t`, and unicode `\u####` where `####` is a four digit hexadecimal number

    - ↳ Keywords in the schema include `"minLength"`, `"maxLength"`, `"pattern"` (for regex), and `"format"` (for specifying specific types of strings, such as `"datetime"`)
  - `"number"`, a signed decimal number optionally with exponential e/E notation
    - ↳ Leading zeros not allowed, except one is required before a decimal if integer part is 0
    - ↳ Leading plus not allowed except in the exponential part
    - ↳ In the schema, one can also use the more restrictive `"integer"`
    - ↳ Keywords in the schema include `"minimum"`, `"maximum"`, `"exclusiveMinimum"`, and `"exclusiveMaximum"`
  - `"object"`, an unordered collection of name/value pairs `{"field1": value1, "field2": value2}`
    - ↳ In the schema, information about its contents are given by a nested schema under keyword `"properties"`
  - `"array"`, an ordered list of zero or more values of any types `[value1, value2]`
    - ↳ For arbitrary-length arrays where each item must meet the same schema, specify this schema using the keyword `"items"`
    - ↳ To validate the first n values (if present) against different schemas, specify an array of schemas using `"prefixItems"`
    - ↳ Use `"minItems"` and `"maxItems"` to validate the length of the array
  - `"boolean"`, which is either `true` or `false`
  - `null`

- A sample JSON file is:

```
{
    "name": "James",
    "classes": [ {"Name": "CSE480", "Students": 135},
                 {"Name": "CSE498", "Students": 178} ],
    "Employed": true
}
```

- A sample JSON schema is:

```
{
    "title": "my_title",
    "description": "my_description",
    "type": "object"
    "properties": {
        "field1": {
            "description": "description1",
            "type": "type1"
        },
        "field2": {
            "description": "description2",
            "type": "type2"
        }
    },
    "required": ["field1", "field2"]
}
```

1.3 XML File

- XML stands for eXtensible Markup Language
  - ↳ HTML is a subset of XML
  - ↳ XML is often used in web services, inter-database communication, text document formatting, and configuration files

- XML also stores data in a semi-structured hierarchical structure under a single root element
- Each element consists of a start tag `<tag_name>`, optional content (without quotes), and an end tag `</tag_name>`, or an empty tag `<tag_name/>` can be used
- Each start or empty tag can also contain attributes `<tag_name attr_name="value">`
    - Only one value may be specified for an attribute, so if multiple are needed, often space-delimited strings are used
- An XML schema uses the following format:

```
<xsd:schema xmlns:xsd="schema_link">
<xsd:annotation>
  <xsd:documentation xml:lang="en">
    Description Here
  </xsd:documentation>
</xsd:annotation>
<xsd:element name="name1" type="myType1"/>
<xsd:complexType name="myType1">
  <xsd:sequence>
    <xsd:element name="name2" type="xsd:type1"/>
    <xsd:element name="name3" type="xsd:type2"/>
  </xsd:sequence>
</xsd:complexType>
</xsd:schema>
```

    - `xsd` specifies the namespace; it is also common to use `xs` instead (in which case the first line of the schema would be changed as well)
- The built-in simple types are `"xsd:string"`, `"xsd:decimal"`, `"xsd:integer"`, `"xsd:boolean"`, `"xsd:date"`, `"xsd:time"`, `"xsd:dateTime"`, and `"xsd:duration"`
- The number of times a tag occurs can be specified with `minOccurs="#"` and `maxOccurs="#"`
    - Use `"unbounded"` for unlimited number of occurrences
    - By default, `minOccurs="1"` and `maxOccurs="1"`, so that the tag is mandatory and can appear only once
- Within a complex type, an attribute can be specified via `<xsd:attribute name="attr_name" type="xsd:type"/>`
- More detailed restrictions can be placed on a xsd type by defining a simple type, such as:

```
<xsd:simpleType name="myType">
  <xsd:restriction base="xsd:integer">
    <xsd:minInclusive value="0"/>
    <xsd:maxInclusive value="120"/>
  </xsd:restriction>
</xsd:simpleType>
```

- Rather than having complex types be named (and then defined later), they can be embedded directly inside the <xsd:element> tag:

```
<xsd:element name="name">
  <xsd:complexType>
    ...
  </xsd:complexType>
</xsd:element>
```

- Alternatively, all elements and attributes can be defined at the beginning without any structure, and then the xsd element tags can use `ref="element_name"` to reference it

2. Database Concepts

2.1 Introduction

- A Database Management System (DBSM) is a software used to store and interact with a database, such as SQLite or DB2
  - ↳ Provides a single interface for interacting with files of potentially different types
  - ↳ Creates a level of abstraction between the data and the application, so that changes to the file structures do not require code changes
  - ↳ Adds the ability to enforce security and permissions for reading/modifying data
- A database instance is the actual data in a database in a moment of time, whereas a database schema is a specification for the type of data that may be in the database
- Well-formed means the data conforms to the file format, whereas valid is a stronger notion meaning that thee data conforms to the schema

2.2 Methods of Organizing Databases

- A hierarchical database uses trees to store one-to-many relationships between entities
  - ↳ For example, a department could have children that are employees, which in turn have children that are project history
  - ↳ This is a fast and easy to understand structure
  - ↳ It requires that programs can traverse the tree in order to locate leaf nodes, which makes it inflexible to structure changes
  - ↳ It is not possible to represent many-to-many relationships without introducing redundancy
- A network database is an extension of hierarchical databases to allow for many-to-many relationships by allowing an entity to have multiple parents
  - ↳ This is also fast, but more complex to understand
  - ↳ It is also inflexible to structure changes
- A relational database stores data in tables (relations) which have columns (attributes) and rows (tuples)
  - ↳ Has mathematical foundation in set theory and relational algebra
  - ↳ Data redundancy is actively reduced via normalization, which reduces storage and helps to avoid anomalies which lead to inconsistencies
  - ↳ It is self-documenting, easy to understand, and allows for restructuring
  - ↳ It is slower since relationships between entities are not enforced by the database structure inherently
  - ↳ Objects present in object-oriented code may need to be taken apart before being stored in the database
- An object-oriented database allows for storage of objects, sometimes based on XML or JSON models
- A NoSQL database describes a mix of organizations, some similar to hierarchical and network organizations, boasting faster performance
  - ↳ This comes at the cost of data consistency – it can only guarantee *eventual* consistency

2.3 Relational Databases

- A relational database stores data in tables, called relations
    - ↳ The rows of a relation are called tuples
    - ↳ The columns of a relation are called attributes
- A relation may have a relation schema, which consists of a name for the relation and a list of its attributes, optionally with their corresponding domains, or sets of possible values
    - ↳ The database is therefore the collection of all relations, whereas the database schema is the collection of all relation schemas
- All values in a relation should be atomic domains, meaning that they contain an indivisible unit of information
    - ↳ A relation is in First Normal Form if all of its domains are atomic (we assume this is the case for all relations)
    - ↳ Sets of names or identification numbers like "CSE101" are not atomic (this should be represented with two separate attributes)
    - ↳ A list of values is not atomic, and should be represented using an additional relation instead
    - ↳ Whether a domain is atomic or not may depend on context
    - ↳ Non-atomic domains complicate storage, queries, and data validation, and also encourage redundant storage of data
- A key is a column or set of columns that uniquely identifies rows in the relation
    - ↳ Every relation must have a key, since the rows in a relation do not have an inherent order and thus must be distinguishable
    - ↳ A key cannot be changed under normal circumstances and cannot have null values
    - ↳ From a set of candidate keys, a single primary key is selected to use, and the rest are called alternate keys
    - ↳ A natural key is a key that is an obvious choice, like student ID
    - ↳ A artificial key or surrogate key is one automatically created by the DBMS when one is not specified (or cannot be created from the existing columns)
        - ↳ In SQLite, a surrogate key called `rowid` is automatically created when an integer primary key is not specified
        - ↳ In DB2, either an identity column is generated automatically, or values can be specified manually by the user with a sequence object (obtaining the next value with `NEXTVAL FOR sequence-object` and the previous value with `PREVVAL FOR sequence-object`)

3. Distributed Databases

3.1 Introduction

- A distributed database stores information on many nodes or virtual machines, run by a Distributed Database Management System (DDBMS)
  - ↳ This is in comparison to a centralized database, in which a single datacenter serves all queries
  - ↳ The multiple datacenters have communications channels to facilitate the transfer of data
  - ↳ Distributed databases allow for greater accessibility, redundancy in case of failure, ability to grow/change without interrupting service, and greater parallelism/concurrency
- A homogeneous distributed database has identical software (OS, DBMS, and schema) used at all sites, whereas a heterogeneous distributed database uses different softwares
  - ↳ Homogeneous distributed databases are easier to treat as a centralized database conceptually
  - ↳ Optimizations used in one datacenter of a homogeneous database can be used at all others, but this is not true for heterogeneous databases
  - ↳ Cooperation between datacenters is easier in the homogeneous case
  - ↳ Heterogeneous databases allow for more diversity of optimization and less reliance on a particular technology
- The CAP Theorem (or Brewer's Theorem) states that a distributed database can only achieve two of the following three guarantees:

  C: Consistency. The same information is seen by all clients regardless of the node accessed; all successful writes should be reflected across all nodes
  - ↳ This is a different consistency then in the ACID properties
  - ↳ A write can only be deemed successful after the information has been successfully propagated

  A: Availability. The database should be kept available and responsive as often as possible; any working nodes must return a valid response for any request

  P: Partition Tolerance. The cluster should continue to work in spite of any partitions; if one part of the database is unavailable, the other parts should remain unaffected
  - ↳ A partition is a communications break within a distributed database; either a lost or delayed connection between two nodes
- In practice, partitions are unavoidable, so partition tolerance must always be included. Thus, the CAP theorem is stating a tradeoff between consistency and availability
  - The PACELC expands upon this by saying that in the absence of a partition, there is a tradeoff between latency and consistency

3.2 Replication and Fragmentation

- Replication is the process of keeping copies of the same data at different sites
  - ↳ This increases availability and safety of the data, but also increases difficulty of data modifications, consistency, and concurrency control
  - ↳ Replication can be complete (the entire database is replicated) or partial (some of the database is replicated)
  - ↳ When no replication is used, the database is partitioned
- For partial replication or partitioned databases, the following strategies can be used to determine which sites store which data:
  - Horizontal partitioning/fragmentation, where a table is partitioned by records. An example is a CSE server holding CSE students and a MTH server holding MTH students

  - Vertical partitioning/fragmentation, where a table is partitioned by columns (i.e., a decomposition is formed). An example is a CSE server holding capstone data and an MSU server holding tuition data
  - Hybrid partitioning/fragmentation, where a mix of both is used

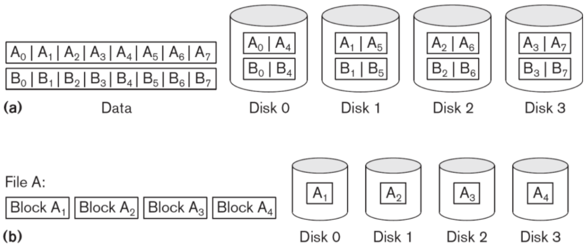
- For horizontal fragmentation, the following strategies can be used:
  - Primary Horizontal Fragmentation, where rows are selected for each fragment using a simple predicate (equality, inequality, or boolean operators)
  - Derived Horizontal Fragmentation, where rows are selected for each fragment based on a particular foreign key

3.3 Locking and Committing Protocol

- A transaction only requiring/affecting data at one local database is a local transaction, while those requiring multiple sites is a global transaction
  - ↳ Local transactions are easy to carry out and commit, while global transactions require more machinery
- For global transactions, the following strategies exist of granting locks:
  - ↳ A primary site can be specified when complete replication is used, and all locking must be approved by this site
  - ↳ This is unreliable, but can be improved by specifying a backup site which immediately carries on with transactions if the primary site fails
  - ↳ This can be extended when complete replication is not used by specifying a primary copy/backup copy for each database item individually
  - ↳ Backup sites/copies can also be determined by an election upon failure
  - ↳ Instead of having primary copies, a vote could be commenced each time a lock is requested, although this can drastically increase network traffic
- For committing, a two-phased commit proceeds as follows:
  - ↳ Commit-Request Phase: the coordinator informs the cohort (sites involved in the commit) of the required changes, and each responds with "yes" or "no"
  - ↳ Commit Phase: the coordinator decides whether to commit or rollback based on responses received and sends the appropriate message
  - ↳ Two-phased commits are a blocking protocol; if the coordinator fails, all sites which responded "yes" are stuck in waiting
- A three-phased commit improves this by adding more communications (but this also increases latency):
  - ↳ Commit-Request Phase: Same as two-phased commit; any failures at this stage result in the transaction aborting (either upon timeout, or because the coordinator issues this)
  - ↳ Prepare-to-Commit Phase: The cohort is notified that they should prepare to commit, and they send an acknowledgment to the coordinator
  - ↳ Commit Phase: Once all acknowledgments are received, a commit message is sent
  - ↳ If the coordinator fails, a new one can be elected, which can attempt to determine when the original coordinator failed and where to resume

3.4 Redundant Array of Inexpensive/Independent Disks (RAID)

- Redundant Array of Inexpensive/Independent Disks (RAID) is a combination of software and hardware meant to improve disk performance
    - ↳ Instead of distributing data across sites, as in the previous conversation, data is distributed across disks to allow for parallelism
- Data Striping is the process of breaking up contiguous data normally contained in a single disk into multiple, either by byte or by block, to allow for parallelism during reads/writes

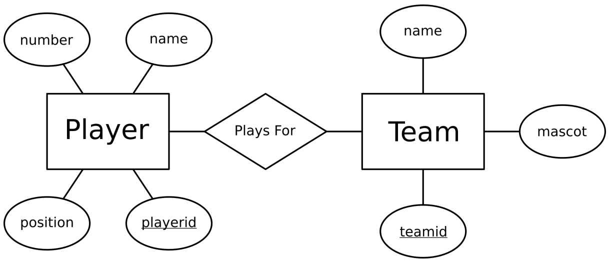


- Mirroring/shadowing is often used, which is the process of replicating data to multiple disks (adding redundancy)
    - ↳ The mean-time-to-failure of a hard drive is 20 years, so if many disks are being used, disk failures can become common
    - ↳ Error-correcting codes can be used to detect small failures in disks
- The following RAID levels categorize combinations of the above:
    - ↳ RAID Level 0, where striping is used without any mirroring/shadowing (popular choice for efficiency, but data is not protected)
    - ↳ RAID Level 1, where the data is mirrored in full on one or more other disks (used for critical applications like logs)
    - ↳ RAID Level 2, where data is striped at the bit level using Hamming Codes for error correction (rarely used due to complexity and inefficiency)
    - ↳ RAID Level 3, where data is striped at the byte level, and a dedicated parity disk stores parity information (less used due to single parity disk being a bottleneck)
    - ↳ RAID Level 4, where data is striped at block level with a dedicated parity disk (less used also due to bottleneck)
    - ↳ RAID Level 5, where data is striped at block level and parity information is distributed (popular choice for balance between performance and redundancy)
    - ↳ RAID Level 6, where data is striped at block level and two sets of parity information are distributed (protection against two simultaneous disk failures)

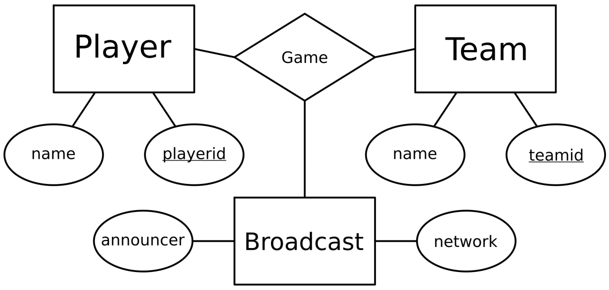
4. Entity-Relationship Model

4.1 Introduction

- The entity-relationship model is a method for describing a system as a database, and is represented using a entity-relationship diagram
  - ↳ An entity is a single distinguishable thing or object, such as a student or a course offering
  - ↳ An entity set is a collection of entities of the same type, such as all students at a university or all course offerings
  - ↳ An attribute is a property of the entities in an entity set, such as student ID or department name (all attributes should be simple, not collections of values)
- In an entity-relationship diagram, rectangles represent entity sets, ovals represent attributes, and diamonds represent relationships between entity sets, such as:



- Relationships can involve more than two entity sets, such as the ternary relationship:



- The value of an entity set is the set of entities belonging to it, and the value of a relationship set is a set of tuples with one component for each related entity set

playerid	name	number	position
0	Jim	5	Forward
1	Jane	17	Shortstop
2	Mary	19	Goalkeeper

Value of Team Entity Set

teamid	name	mascot
4	Detroit	Pistons
9	Los Angeles	Angels
15	Seattle	Sounders

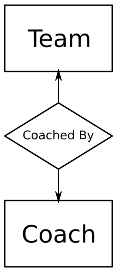
Value of Teams Entity Set

playerid	teamid
0	4
1	9
2	15

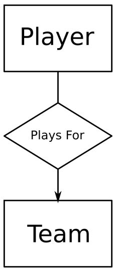
Value of Plays For Relationship Set

- For a binary relationship between A and B, the following mapping cardinalities or multiplicities are possible:
  - ↳ One-to-one, where each entity of A is associated with at most one entity of B and vice versa
  - ↳ One-to-many, where each entity of B is associated to at most one entity of A (but any entity of A can be associated to any number of entities of B)
  - ↳ Many-to-one, where each entity of A is associated to at most one entity of B (but any entity of B can be associated to any number of entities of A)
  - ↳ Many-to-many, where each entity of A is associated to any number of entities in B and vice versa

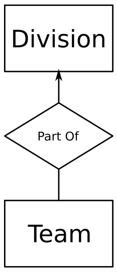
- If one entity is related to exactly one of another entity, this is represent with a rounded arrow



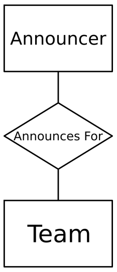
One-to-One



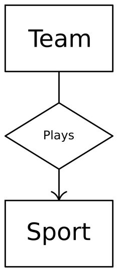
Many-to-One



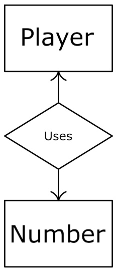
One-to-Many



Many-to-Many



Many-to-Exactly-One



Exactly-One-to-Exactly-One

- Attributes can be present on a relationship:

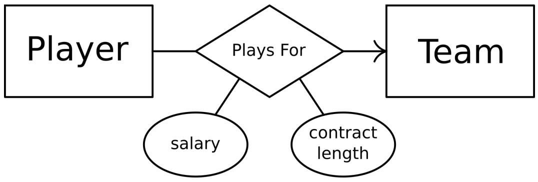


Diagram with Attributes on Relationship

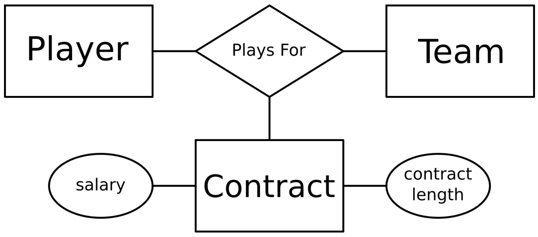
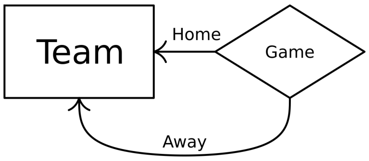


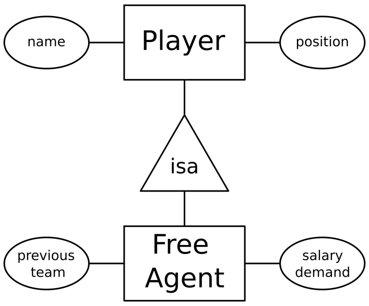
Diagram without Attributes on Relationship

- Edges are sometimes labeled with roles to disambiguate the case where an entity set appears multiple times in a relationship



4.2 Subclasses

- A subclass is a special case of an entity set with more properties (either attributes or relationships)
  ↳ Note that no multiple inheritance is allowed



- There are multiple ways to implement subclasses:
  (i) Nulls: Use one relation with all possible attributes, and enter NULL for inapplicable attributes.
  (ii) Object-Oriented: Use one relation per subset of subclasses, each with all relevant attributes. Only list an object in one relation.

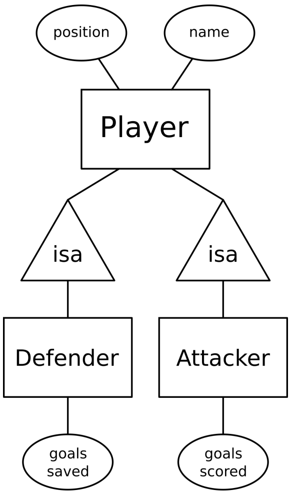
(iii) E/R-Style: Use one relation for each subclass with attributes of that subclass and key attributes. List an object in the subclass relation and all superclass relations.

- The E/R-Style is generally preferred, but it requires more complicated queries with joins

player

name	position	saved	scored
Jimmy	Midfield	2	4
Jane	Defense	7	NULL
Jessica	Attack	NULL	14
Justin	Bench	NULL	NULL

Null Method



player

name	position
Justin	Bench

attacker and defender

name	position	saved	scored
Jimmy	Midfield	2	4

attacker

name	position	scored
Jessica	Attack	14

defender

name	position	saved
Jane	Defense	7

Object-Oriented Method

player

name	position
Jimmy	Midfield
Jane	Defense
Jessica	Attack
Justin	Bench

defender

name	saved
Jane	7
Jimmy	2

attacker

name	scored
Jessica	14
Jimmy	4

E/R-Style

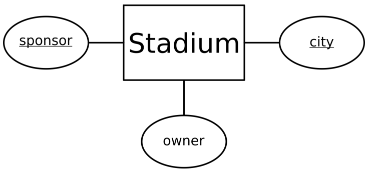
4.3 Keys

- A key is a set of attributes for one entity set so that no two entities agree on all attributes in the key

  ↳ The key uniquely identifies each entity in the entity set

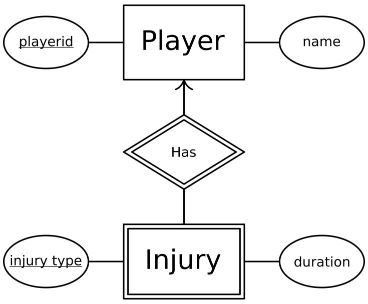
  ↳ Every entity set must have a key, which is denoted with an underline

  ↳ In a "isa" hierarchy, only the root entity set has a key, which must work for all subclasses



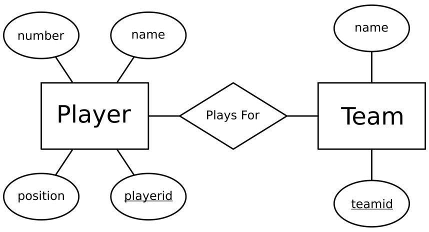
- A weak entity E is one such that the key of other entity sets needs to be included in order to uniquely identify entries, by following at least one many-to-exactly-one relationship from E

  ↳ This is denoted by adding a double outline around the weak identity and supporting relationships

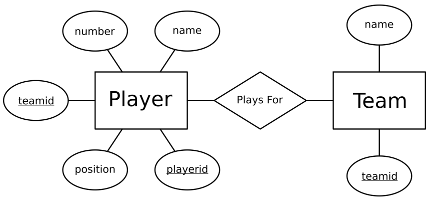


4.4 Design Decisions

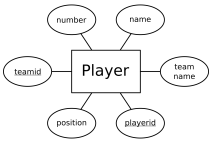
- A database should avoid redundancy, both to limit space requirements and to reduce the risk of inconsistency



Good Example



Bad Example



Bad Example

- An entity set should be used instead of an attribute only when either:
  (i) The quantity is more than just the name of something; it has at least one non-key attribute
  (ii) The quantity is the "many" part in a "many-to-one" or "many-to-many" relationship
  ↳ For example, even if player only had one attribute, it would need to be an entity set since many players belong to a team
  ↳ If team only had a single attribute, it should be an attribute of player rather than an entity set

- Weak entities should not be overused
  ↳ It is better to create unique IDs for entity sets (e.g., social security numbers and automobile VINs)
  ↳ Weak entities are only needed when there is no global authority capable of establishing a unique identifier

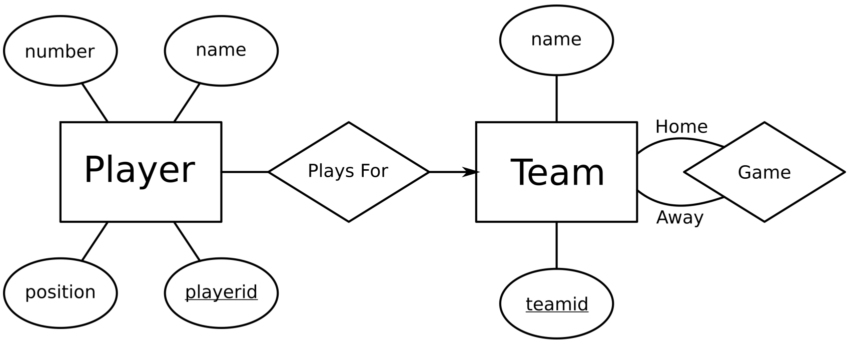
4.5 Converting E/R Diagrams to Relations

- When converting an E/R diagram to relations/tables, each entity set becomes a relation with the corresponding attributes

- Each relationship becomes a relation whose attributes are:
  (i) The keys of the connected entity sets
  (ii) Any attributes of the relationship itself

- All attributes needed to form the key of a weak entity set must be included in the relation

- In a many-to-one relationship, the table for the relationship can be combined with the "many" table



Relations:

- Player(playerid , name, number, position, teamid)
  - ↳ This is the result of merging Player(playerid , name, number, position) and PlaysFor(playerid , teamid)
- Team(teamid, name)
- Game(home, away)

5. Relational Database Design

5.1 Introduction

- While the E/R-model provides an intuitive means of designing a database, there are also more systematic/mathematical ways to accomplish this task
- Consider all the data in a database stored in a single table, with null values and redundancy. Decompositions of this table are made when it is divided into smaller relations/entity sets. All decompositions should be lossless, so that the original table could be recovered via natural joins on the subrelations. The process of iteratively creating decompositions ultimately yields a database design
- Redundancy can occur when not enough decompositions are specified, i.e., the relations contain too much information and should be split into separate entity sets
  - ↳ This can lead to anomalies, which are problems that occur specifically when too much information is contained a single relation
  - ↳ Update anomalies occur when redundant information is updated in one location but not in other locations
  - ↳ Deletion anomalies occur when a deletion causes more information to be deleted than desired (for example, deleting information about a project grade when a student drops the class)
  - ↳ Insert anomalies occur when data cannot be inserted into a relation because another piece of data is missing
- We also assume that all relations are in First Normal Form (all domains are atomic)

5.2 Functional Dependency

- A set of attributes Y is functionally dependent on a set of attributes X in relation R, denoted $X \to Y$, if any rows agreeing on the values in X also agree on the values in Y
  - ↳ $X \to Y$ means the values of Y are entirely determined by the values in X, in relation R
  - ↳ Notationally, X, Y, Z are used to denote sets of attributes, while A, B, C are used to denote individual attributes. Also, $\{A, B, C\}$ is typically denoted ABC
  - ↳ $X \to A_1 A_2 \cdots A_n$ is equivalent to $X \to A_i$ for all $1 \leq i \leq n$ (this is the splitting/combining rule)
  - ↳ A functional dependency $X \to Y$ is trivial if $Y \subset X$
- K is a superkey for relation R if $K \to R$, and K is a key if it is minimal (i.e., no proper subset of K is also a superkey)
  - ↳ Keys can be determined by beginning with all of R and then iteratively examining its subsets, or through deduction based on the functional dependencies in R

5.3 Closure of Functional Dependencies

- Given a set F of functional dependencies, the closure F^+ of F is the set of all functional dependencies logically implied by those in F
- Armstrong's axioms provide a sound and complete logical system for functional dependencies, and thus can be used to generate F^+. For attribute sets $\alpha, \beta, \gamma, \delta$:

  (i) Reflexivity: If $\beta \subseteq \alpha$, then $\alpha \to \beta$

  (ii) Augmentation: If $\alpha \to \beta$, then $\gamma\alpha \to \gamma\beta$

  (iii) Transitivity: If $\alpha \to \beta$ and $\beta \to \gamma$, then $\alpha \to \gamma$
- Several additional rules can be derived from these:
  - ↳ Union Rule: $\alpha \to \beta$ and $\alpha \to \gamma$ implies $\alpha \to \beta\gamma$

- ↳ Decomposition Rule: $\alpha \to \beta\gamma$ implies $\alpha \to \beta$ and $\alpha \to \gamma$
- ↳ Pseudotransitivity: $\alpha \to \beta$ and $\beta\gamma \to \delta$ implies $\alpha\gamma \to \delta$

- We can generate F^+ as follows:

Algorithm 1: Procedure to Compute F^+

```
F^+ ← F
apply reflexivity      // generates all trivial dependencies
repeat
    foreach functional dependency $f \in F^+$ do
        apply the augmentation rule on $f$
        add the resulting functional dependencies to $F^+$
    end
    foreach pair of functional dependencies $f_1, f_2 \in F^+$ do
        if $f_1, f_2$ can be combined using transitivity then
            add the resulting functional dependencies to $F^+$
        end
    end
until $F^+$ does not change
```

- The closure α^+ of a set of attributes α under F is the set of attributes that are functionally determined by α under F
  - ↳ One method is to calculate F^+, but this can be expensive
  - ↳ Another method essentially applies transitivity repeatedly:

Algorithm 2: Procedure to Compute α^+

```
$\alpha^+ \leftarrow \alpha$
repeat
    foreach functional dependency $(\beta \to \gamma) \in F$ do
        if $\beta \subseteq \alpha^+$ then
            $\alpha^+ \leftarrow \alpha^+ \cup \gamma$
        end
    end
until $\alpha^+$ does not change
```

- α is a superkey iff $\alpha^+ = R$
- $\alpha \to \beta$ iff $\beta \subseteq \alpha^+$

5.4 Boyce-Codd Normal Form (BCNF)

- A decomposition of $R = R_1 \cup R_2$ is a lossless join iff $R_1 \cap R_2$ is a superkey for either R_1 or R_2
  - ↳ This implies either $R_1 \cap R_2 \to R_1$ or $R_1 \cap R_2 \to R_2$
  - ↳ Note implicitly that we have required all attributes be present in either R_1 or R_2
- The Boyce-Codd Normal Form (BCNF) ensures that no redundancy arising from functional dependencies are possible
  - ↳ R is in BCNF iff for any nontrivial functional dependency $\alpha \to \beta$ in F^+, we have that α is a superkey for R
  - ↳ For example, $R = \{\text{team\_id}, \text{team\_name}, \text{trophies}, \text{player\_id}, \text{player\_name}, \text{player\_position}\}$ with the obvious functional dependencies is not in BCNF, since team_id is not a superkey of R but functionally determines team_name

- To check whether R is in BCNF, it suffices to check that none of the functional dependencies in F violate the rule, rather than checking all of F^+
    - ↳ However, this is not true after R is decomposed, say into R_1 and R_2. For example, let $R = \{A, B, C, D\}$, $R_1 = \{A, C, D\}$, and $R_2 = \{A, B\}$, with the functional dependencies $A \rightarrow B$ and $B \rightarrow C$. Although neither of these are present in R_1 or R_2, transitivity yields $A \rightarrow C$, which violates BCNF for R_1
    - ↳ Instead, compute α^+ for each subset $\alpha \subset R_i$, and confirm that either $R_i \subset \alpha^+$ or $\alpha^+ \cap R_i = \alpha$

Algorithm 3: BCNF Decomposition Algorithm

```
result = {$R$}
**while** *there is a schema $R_i \in$ results that is not in BCNF* **do**
    let $\alpha \rightarrow \beta$ be a nontrivial functional dependency holding on $R_i$ such
      that $R_i \not\subset \alpha^+$ and $\alpha \cap \beta = \varnothing$
    replace $R_i$ with $(R_i - \beta)$ and $(\alpha, \beta)$ in results
**end**
```

- ↳ The relation is decomposed into one containing just the attributes involved in the functional dependency and one containing all attributes except those functionally determined by α
- ↳ When choosing $\alpha \rightarrow \beta$, it is best to let $\beta = \alpha^+$, so that fewer decompositions are formed

5.5 Canonical Cover

- During any update, in theory, the database system should ensure that functional dependencies are satisfied
    - ↳ In practice, this is not done since it is expensive and generally difficult
    - ↳ However, by developing theory to make this more efficient, we can develop forms (such as Third Normal Form) in which this is possible
- Extraneous attributes of functional dependency $\alpha \rightarrow \beta$ are those that could be removed without changing the closure set F^+ of all functional dependencies
    - ↳ $A \in \alpha$ is extraneous if F logically implies $(F - \{\alpha \rightarrow \beta\}) \cup \{(\alpha - A) \rightarrow \beta\}$
    - ↳ $A \in \beta$ is extraneous if $(F - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - A)\}$ logically implies F
- The reverse direction for each of the above is always true
- To check for extraneous attributes, either:
    - ↳ If $A \in \alpha$, then compute the closure of $\alpha - A$ under F, and determine if all attributes of β are contained in it
    - ↳ If $A \in \beta$, then compute the closure of α under $(F - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - A)\}$, and determine if it includes A
- A canonical cover F_c for F is a set of dependencies equivalent to F (so that each logically implies the other) with the minimality properties:
    - No functional dependency in F_c contains an extraneous attribute
    - Each left side of a functional dependency in F_c is unique (i.e., there does not exist $\alpha_1 \rightarrow \beta_1$ and $\alpha_2 \rightarrow \beta_2$ with $\alpha_1 = \alpha_2$)
- There may exist multiple canonical covers (it is not unique)
- To compute a canonical cover, begin with $F_c = F$ then iteratively use the union rule to combine dependencies with the same left side, and remove extraneous attributes *computed using the current value of F_c*

5.6 Third Normal Form (3NF)

- Another desirable quality of database designs (other than avoiding redundancy with BCNF) is ensuring dependency preservation
    - ↳ Dependency preservation states that all functional dependencies in F^+ should be able to be checked solely by examining the quantities within each relation, *without* needing to perform joins
    - ↳ Joins can be expensive, and aside from that it is more clear and natural to have functional dependencies represented within a single relation
    - ↳ Formally, suppose R is decomposed into relations $R_1, \dots, R_n$. Define F_i to be the restriction of F to R_i, which is the set of all functional dependencies in F^+ including only attributes in R_i. The functional dependencies $F' = F_1 \cup \cdots \cup F_n$ are those that can be efficiently verified without joins. If $(F')^+ = F^+$, then the decomposition satisfies dependency preservation, since verifying dependencies in F' is sufficient to confirm that all those in F hold.
- Third Normal Form (3NF) is a weaker form of BCNF which allows enough flexibility for a lossless dependency-preserving decomposition to exist; for every nontrivial $\alpha \to \beta$, either α is a superkey for R as in BCNF, or each attribute in $\beta - \alpha$ belongs to some key in R
    - ↳ It may be that each of the different attributes in $\beta - \alpha$ belong to a different possible key for R; they do not all need to belong to the same key

Algorithm 4: 3NF Dependency-Preserving Lossless Decomposition Algorithm

```
compute canonical cover $F_c$ of $F$
foreach $\alpha \to \beta$ in $F_c$ do
    add relation $\alpha\beta$
end
if no schema generated contains a key for $R$ then
    add relation containing just a key for $R$
end
/* Optionally remove redundant relations */
repeat
    if any schema $R_j$ is contained in another schema $R_k$ then
        delete $R_j$
    end
until no more relations can be removed
```

- ↳ This algorithm ensures dependency preservation by explicitly building a relation for each functional dependency in the canonical cover, which can lead to a high amount of redundancy and unnecessary decompositions
- ↳ However, this algorithm is a proof by construction that 3NF is sufficiently relaxed to allow for dependency preservation
- ↳ The algorithm is clearly dependency preserving since every dependency in the canonical cover is represented in a single relation. To see that it generates relations R in 3NF, consider nontrivial $\gamma \to \delta$ that holds on R. Without loss of generality, we can let δ be a single attribute, so that we can consider nontrivial $\gamma \to B$. Let $\alpha \to \beta$ be the functional dependency that generated R in the above algorithm. Since $R = \alpha\beta$, either B is in α or β, so one of the following is true:
    - $B \in \alpha$ and $B \in \beta$. Then B is extraneous, so that $\alpha \to \beta$ would not be in F_c, a contradiction.
    - $B \in \alpha$ and $B \notin \beta$. α is a key since $\alpha \to \beta$ and $R = \alpha\beta$, so $\gamma \to B$ does not violate 3NF.
    - $B \in \beta$ and $B \notin \alpha$. Either:
        - γ is a superkey, so that $\gamma \to B$ does not violate 3NF.

- γ is not a superkey. We claim that B is extraneous in $\alpha \to \beta$, a contradiction since $\alpha \to \beta$ is in F_c. From $F' = (F - \{\alpha \to \beta\}) \cup \{\alpha \to (\beta - B)\}$, we must show that $\alpha \to B$. $\gamma \to B$ is in F_c and thus is derivable from F, and this derivation could not have used $\alpha \to \beta$ since γ is not a superkey and thus $\alpha \not\subset \gamma^+$. Therefore, $\gamma \to B$ is logically implied by F'. Rewrite $\alpha \to (\beta - B)$ as $\alpha \to (\alpha\beta - B)$. Since $\gamma \subset \alpha\beta - B$ (noting that $B \notin \gamma$), we have $\alpha \to \gamma$, and thus $\alpha \to B$.

6. SQL

6.1 Introduction to SQL

- Databases typically use restricted languages such as SQL
  - ↳ This is easier to optimize and faster to write
  - ↳ Often they are not Turing-complete, although technically SQL is
- SQL offers both querying and a component for data-definition (describing database schemas)
- SQLite is a database engine which can be used in Python, while DB2 is a database engine provided by IBM used with COBOL and CICS
- The four basic operations for persistent storage are create, read, update, and delete (acronym CRUD)
- SQL statements can more specifically be divided into the categories:
  - ↳ Data Definition Language (DDL), which build/modify the structure or schema of the database elements: `CREATE`, `ALTER`, `DROP`
  - ↳ Data Query Language (DQL), which reads data: `SELECT`
  - ↳ Data Manipulation Language (DML), which manipulate data within relations: `INSERT`, `UPDATE`, `DELETE`
  - ↳ Data Control Language (DCL), which grants or revokes access privileges
  - ↳ Transaction Control Language (TCL), which controls transactions: `COMMIT`, `ROLLBACK`

6.2 SQL Style

- Semicolons should terminate each statement
- SQL keywords should be in all capitals
- Whitespace is only relevant in strings, although good practices are shown in the examples below
- When using DB2, whitespace is typically formatted very diligently, as with COBOL; an example of a well-styled query is:

```sql
SELECT DEPT_DESC AS "DEPARTMENT"
       COUNT(*)  AS "NUMBER OF EMPLOYEES"
  FROM EMPLOYEE_RECORDS R
       JOIN DEPT_DESC D
         ON R.DEPT_NUM = D.DEPT_NUM_KEY
 WHERE R.EMPLOYEE_SALARY > 50000
   AND R.DEPT_NUM LIKE '5%'
 GROUP BY R.DEPT_NUM
HAVING COUNT(*) > 20
 ORDER BY COUNT(*) DESC, DEPT_DESC
;
```

6.3 SQLite Datatypes

- SQLite offers the following datatypes:
  - – `INTEGER`, a signed integer up to 8 bytes long
  - – `REAL`, a signed float using decimal or exponential notation (using capital E)
  - – `TEXT`, a string of any length delimited by single quotes
    - ↳ Single quotes are escaped by including two of them

        - ↳ Note that double quotes are used for alias names
        - ↳ Other databases limit the length of strings, but SQLite does not
    - BLOB, used to hold arbitrary binary data, such as images or videos
        - ↳ This has no storage limits, although storing file paths is often easier

- Any datatype may also be NULL
    - ↳ It is used to represent an absence of information or an inapplicable value
    - ↳ It should not be used to represent empty strings, false, or 0
- Dates and times can be represented as TEXT (ISO8601 strings, "YYYY-MM-DD HH:MM:SS.SSS"), REAL (Julian day numbers), or INTEGER (Unix time); see section 6.22
- Other types not in SQLite include Memo (65536 characters), Currency (15 whole digits and 4 decimal places), Yes/No (Microsoft Access), and Enum
- One way of working with other types in SQLite is to use adapters/converters
    - ↳ An adapter is a function for converting a Python object into one of SQL's supported types, and is registered with `sqlite3.register_adapter(python_type, adapter)`
    - ↳ A converter is a function converting a bytes object back into a desired Python object, and is registered with `sqlite3.register_converter(sql_name, converter)`
    - ↳ Note that these are registered with the module, not the connection
    - ↳ To instruct SQLite to convert custom types during insertion, pass `detect_types=sqlite3.PARSE_DECLTYPES` when forming the connection
    - ↳ To apply converter upon selection, use `SELECT col AS "col [type]" FROM table;` and specify `detect_types=sqlite3.PARSE_COLNAMES` when forming the connection
    - ↳ Both options can be specified by using `detect_types=sqlite3.PARSE_DECLTYPES | sqlite3.PARSE_COLNAMES`

```python
def adapt_color(color):
    return f"{color.r};{color.g};{color.b}"
def convert_color(bytestring):
    as_str = bytestring.decode("ascii")
    r, g, b = [float(x) for x in as_str.split(';')]
    return Color(r,g,b)
sqlite3.register_adapter(Color, adapt_color)
sqlite3.register_converter("COLOR", convert_color)
c = Color(24, 69, 59)
conn = sqlite3.connect(":memory:", detect_types = sqlite3.PARSE_DECLTYPES | sqlite3.PARSE_COLNAMES)
conn.execute("CREATE TABLE my_colors (col COLOR);")
conn.execute("INSERT INTO my_colors VALUES (?);", c)
res = conn.execute('SELECT col AS "col [COLOR]" FROM my_colors;')
row = next(res)
```

6.4 DB2 Datatypes

- Character:
    - CHAR(n): A fixed-length character string of length n (shorter strings are padded with trailing spaces)
    - VARCHAR(n): A variable-length character string of maximum length n
    - GRAPHIC(n): A fixed-length character string with double-byte characters (allowing for special characters like in Chinese and Japanese)

  - `VARGRAPHIC(n)`: A variable-length character string of maximum length n with double-byte characters
  - `CLOB`: A character large object with maximum length approximately 2 GB (useful for length documents)
  - `DBCLOB`: A double-byte character large object with maximum length approximately 2 GB (1073741823 double-byte characters)
  - `XML`: A variable-length string containing an XML document

- Numeric:
  - `SMALLINT`: An integer between −32768 and 32767
  - `INTEGER` or `INT`: An integer between approximately ±2 billion
  - `BIGINT`: An integer between approximately ±9 quintillion
  - `DECIMAL(p,s)` or `NUMERIC(p,s)`: A decimal with precision `p` (the total number of digits) and scale `s` (number of decimal places)
    - ↳ This is also used for integers where there is an expectation on the number of digits, such as a salary
  - `REAL`: A single precision floating-point number (32 bits)
  - `DOUBLE` or `FLOAT`: A double precision floating-point number (64 bits)
  - `DECFLOAT`: A number with the decimal stored in the value itself, rather than via a sign/exponent/mantissa

- Temporal:
  - `DATE`: A date in `YYYY-MM-DD` format
  - `TIME`: A time in `HH.MM.SS` format
  - `TIMESTAMP`: A date and time with format `YYYY-MM-DD HH.MM.SS.mmmmmm` where microseconds are included

- Binary Strings:
  - `BINARY(n)`: A binary string with fixed-length specified between 1 and 255
  - `VARBINARY(n)`: A variable-length binary string with maximum length specified between 1 and 32704
  - `BLOB`: A binary large object with maximum length approximately 2 GB, useful for images, videos, JSON data, etc.

6.5 Adding Data

- A relation is created via the `CREATE TABLE` statement:

```
CREATE TABLE table_name (
    column1 datatype,
    column2 datatype,
    column3 datatype
);
```

- The datatypes are optional, and are not enforced by SQLite
- A relation can be created from another table/query by:

```
CREATE TABLE new_table AS
SELECT column1, column2
FROM old_table
```

- Records are added to the table via the `INSERT INTO` statement:

```
INSERT INTO table_name (column1, column2, column3)
VALUES (value1a, value2a, value3a),
       (value1b, value2b, value3b);
```

↳ The columns need not be specified if values are added for all columns in order

↳ Unspecified attributes are given value `NULL`

↳ A `SELECT` query can also be used to fill the rows needed for `INSERT INTO`:

```
INSERT INTO valued_customers
SELECT * FROM customers
ORDER BY money_spent DESC
LIMIT 20;
```

6.6 Updating Data

- The `UPDATE` command is used to update one or more records, with syntax `UPDATE table SET col = value WHERE condition`

```
UPDATE guestbook
SET name = 'Mr. ' || name
WHERE gender = 'Male';
```

- Multiple columns can be updated with `col1=value1, col2=value2, ...`
- The syntax `SET (col1, col2, ...) = (value1, value2, ...)` is also possible but not preferred

6.7 Removing Data

- `DELETE FROM` permanently removes records from a table
- `DELETE FROM table` removes all rows in `table`
- `DELETE FROM table WHERE condition` removes rows meeting a condition

6.8 Querying

- Querying a relation is performed with the `SELECT` statement:

```
SELECT column1, column2, column3
FROM table_name;
```

↳ All attributes may be selected with the wildcard character *

↳ Other queries may order or filter by columns; these need not be selected. `SELECT` is essentially just controlling the final output to the user

- In SQLite, the `FROM` clause is not required if selecting a literal

↳ In DB2, it is required, so one can select from the table `SYSIBM.SYSDUMMY1`

- To order the rows by an column (or a set of columns), use the `ORDER BY` statement:

```
SELECT column1, column2, column3
FROM table_name
ORDER BY column1 ASC|DESC, column2 ASC|DESC;
```

  - The order is ascending by default, although `DESC` can be specified to make it descending
  - A number can also be specified in place of a column, where 1 means the first column, 2 means the second, and so on (this is discouraged because changes to the database structure would require code changes)
  - In SQLite, nulls are treated as low values, although this can be modified by specifying `NULLS FIRST|LAST` in the order clause
  - In DB2, nulls are treated as high values

- Rows can be filtered with the `WHERE` clause (the rows are then included iff the condition evaluates to `TRUE`):

```
SELECT column1, column2
FROM table_name
WHERE condition;
```

- `LIMIT n` can be used at the end of the query to only return at most `n` rows
- A `SELECT` statement may select arbitrary expressions, and need not have `FROM`, such as `SELECT 1+2+3+4;`
- `DISTINCT` can be used to select distinct/unique values in a column, with syntax `SELECT DISTINCT col1 col2 ...`
  - Since `DISTINCT` is an operator returning distinct values, it can also be used inside functions, such as `count(DISTINCT col1)`
  - `DISTINCT` can only operate on a single column in SQLite, whereas multiple columns are supported in DB2
- Altogether, a `SELECT` statement may have the following parts, in order: `SELECT`, `FROM`, `WHERE`, `GROUP BY`, `HAVING`, `ORDER BY`, `LIMIT`

6.9 SQL Operators

- Arithmetic operators are `+`, `-`, `*`, `/`, and `%`
- String concatenation is `||`
- Comparison operators are `=`, `!=` or `<>`, `<`, `>`, `>=`, and `<=`
  - All comparisons with `NULL` yield `UNKNOWN`, rather than `TRUE` or `FALSE`
- The `BETWEEN` operator checks whether the column is between two values inclusive, such as `BETWEEN lower AND upper`
- `NOT` can be used to negate a condition, such as `NOT BETWEEN`
- `IS` and `IS NOT` is used to compare the attribute with `NULL`
  - This is the only time `IS` is used, since normal comparisons always yield `UNKNOWN` with `NULL`
- `AND` and `OR` are used to combine conditions
- `LIKE` is used to select strings matching a pattern, such as `LIKE 'prefix%'`
  - `%` is a wildcard matching 0 or more characters
  - `_` is a wildcard matching a single character
  - An escape character can be specified using `LIKE 'ab\%cd\\' escape '\'`, so that this would match the string `'ab%cd\'`
  - Using `%` as the first character can be inefficient since it leads to a full table scan, so it is worthwhile to consider whether alternatives exist
  - In DB2, trailing spaces in fixed-width fields must be accounted for in the pattern
- `IN` is used to select values that are in a list, such as `IN (1,3,5)` (or it can be used with subqueries)

6.10 SQLite Scalar Functions

- `abs(x)` returns the absolute value of `x`
- `lower(x)` and `upper(x)` return copies of a string with the case changed
- `random(x)` returns a random, signed 64 bit integer
- `typeof(x)` returns `"null"`, `"integer"`, `"real"`, `"text"`, or `"blob"`
- `round(x,y)` returns `x` rounded to `y` decimal places
- `trim(x,y)` returns a copy of string `x` with any copies of character `y` removed from either end
- Users can define functions in Python and pass them to SQLite using `conn.create_function(name, num_params, func)`
  - ↳ The function must return precisely one value, which must be a SQLite-supported datatype
  - ↳ Functions can be overloaded by the number of parameters

6.11 DB2 Scalar Functions

- `CONCAT(string_expr1, string_expr2)` concatenates two strings, equivalent to `||`
- `STRIP(string_expr)`, `STRIP(string_expr, location)`, `STRIP(string_expr, location, trim_char)`, `TRIM(string_expr)`, `TRIM(location FROM trim_char)`, `TRIM(location trim_char FROM string_expr)` all remove leading and/or trailing spaces
  - ↳ `location` defaults to `BOTH`, but can be specified as `LEADING`/`L`, `TRAILING`/`T`, or `BOTH`/`B`
  - ↳ `trim_char` defaults to `SPACES`
- `LENGTH(expr)` roughly returns the length of a value, but this depends on the datatype:
  - ↳ For `CHAR` or `VARCHAR` types, it return the number of characters, which is the same as the number of bytes needed to store the string. For fixed-length strings, this is always constant.
  - ↳ For `GRAPHIC`, this return the number of character (half the number of bytes needed to store the string)
  - ↳ For all other datatypes, it returns the number of bytes needed to hold the value
  - ↳ If `expr` is null, then so is `LENGTH(expr)`
- `SUBSTR(string, start_pos, length)` extracts a substring from a character type
  - ↳ `start_pos` is 1-indexed
  - ↳ `length` is optional, and if not specified, the substring extends to the end of the original string
- `POSSTR(source_string, search_string)` searches a string for the location of the first occurrence of a substring (returning 0 if not found, or `NULL` if either argument is `NULL`)
- `UPPER(string)` and `LOWER(string)` convert the case of a character string
- `LPAD(string, length, pad_string)` and `RPAD(string, length, pad_string)` pad a string to a desired length, either by adding characters to the left or right
  - ↳ Strings that are too long are truncated
  - ↳ `pad_string` is optional, and default to `SPACES`. If it contains more than one character, then characters are pulled from the string in order as needed, repeating if necessary
- `DECIMAL(string, precision, scale)` returns a numeric representation of a character string
  - ↳ `precision` and `scale` are each optional. `precision` defaults to a value dependent on the resulting datatype, and `scale` defaults to 0 (no decimal places).
  - ↳ If `precision` or `scale` are not large enough, then an error is thrown (no rounding or truncation are performed)

↳ If string is NULL, then the output is also NULL

- CHAR(numeric_expr) returns a character representation of a string (or NULL if the input is NULL)

  ↳ The string is allocated enough space to hold the largest value of the numeric datatype, a sign, and a decimal place if the type is not an integer. The result is padded on the left with spaces.

- NULLIF(expr1, expr2) returns expr1 unless the two expressions are equal, in which case NULL is returned.

  ↳ If either expression is NULL, then NULL is returned.

- IFNULL(expr1, expr2) returns expr1 unless it is NULL, in which case expr2 is returned

  ↳ The types of expr1 and expr2 must be compatible

- COALESCE(expr1, expr2, ...) returns the first non-null value, or NULL if all of the expressions are NULL (and again the types of the expressions must be compatible)

6.12 SQL Aggregators

- Aggregators take 0 or more values and return some form of summary of those values; they can be used in select statements, but not in WHERE clauses

  ↳ If this is needed, a subquery can be used to select the appropriate value

- In DB2 and many other engines, an ordinary column cannot be selected along with an aggregate value

  ↳ In SQLite, selecting a column along with an aggregate does not through an error (for min and max, it returns the info for one of the rows achieving the minimum or maximum; otherwise, it is information for an arbitrary row)

- min and max return the minimum/maximum value of a column

  ↳ When used in a select statement, the query essentially returns a single row whose value is minimal or maximal in the specified column, so that SELECT col1, min(col2) would return a single row with the minimum of col2 and some corresponding value in col1

- count(col1) returns the number of non-null values in col1

  ↳ count(*) returns the number of rows in the table, including rows all of whose values are NULL

- sum returns the sum of the column, and NULL if all values are NULL

- total returns the sum of the column, where NULL is treated as 0.0 (note that all outputs are floats)

- avg returns the average of a column

- Users can define aggregators in Python and pass them to SQLite using conn.create_aggregate(name, num_params, agg) where agg is a class with three methods:

  ↳ __init__(self), used to initialize the class

  ↳ step(self, value), a method that is called for each value being aggregated

  ↳ finalize(self), a method that returns the value for the aggregate

6.13 Collations

- A collation defines how data is sorted and how equality is checked
- SQLite has three built-in collations:
  - ↳ `BINARY`, the default which compares the binary representation of the two values
  - ↳ `NOCASE`, where uppercase letters are treated like lowercase
  - ↳ `RTRIM`, where trailing whitespace is ignored
- Collations can be specified in two ways:
  - ↳ When creating the table, with `CREATE TABLE name (col type COLLATE collation);`
  - ↳ When selecting, with `SELECT * FROM table ORDER BY col COLLATE collation;`
- Users can define collations in Python and pass them to SQLite using `conn.create_collation(name, collation)` where `collation` is a callable taking two values and returning:
  - ↳ `0` if the values are equal
  - ↳ `-1` if the first value is smaller than the second value
  - ↳ `1` if the first value is greater than the second value
- A collation can be removed with `conn.create_collation(name, None)`

6.14 Case Expression

- A simple case expression acts like a switch statement, such as in:

```
UPDATE table SET col =
CASE expr
  WHEN expr_val_1 THEN val_1
  WHEN expr_val_2 THEN val_2
  ELSE val_3
END;
```

- A searched case expression uses separate conditions for each `WHEN` clause:

```
UPDATE table SET col =
CASE
  WHEN cond_1 THEN val_1
  WHEN cond_2 THEN val_2
  ELSE val_3
END;
```

- If no `ELSE` clause is specified and none of the conditions are matched, then `NULL` is returned

6.15 Column Constraints

- `UNIQUE` is used to specify that the column must contain unique values, such as:

```
CREATE TABLE students (
    msu_id TEXT UNIQUE,
    pid INTEGER UNIQUE,
    first_name TEXT)
);
```

  - ↳ Duplicate `NULL` values are allowed

- `NOT NULL` can be specified to prevent `NULL` values
- Each table can have at most one `PRIMARY KEY`, which is implicitly `NOT NULL` and `UNIQUE`
    - ↳ It is used to specify to make joins easier and specifies how the database should internally organize itself (for optimized queries)
    - ↳ A composite primary key can be specified with `PRIMARY KEY (column1, column2)`
    - ↳ `INTEGER PRIMARY KEY` will automatically autofill with a unique value if one isn't provided
    - ↳ `INTEGER PRIMARY KEY AUTOINCREMENT` causes the keys to be picked in order with a step size of 1 (although this is expensive and discouraged)

6.16 Table Constraints

- At the end of the list of columns in a `CREATE TABLE` statement, one can go on to list table constraints
- `FOREIGN KEY` is used to indicate that a column references the primary key of another table, with syntax `FOREIGN KEY (col_in_table) REFERENCES table(id)`
    - ↳ The foreign key may be `NULL`
    - ↳ Foreign key checking is expensive, so it is only performed if turned on via `PRAGMA foreign_keys = ON;`
    - ↳ If multiple statements are part of the same transaction, it is possible that the check only happens at the end of the transaction
- Any arbitrary constraint can be made with `CHECK(condition)`, which is enforced upon insertion or updating
- `FOREIGN KEY` can also be expressed with `CHECK` as:

```sql
CHECK (col_in_table IS NULL OR EXISTS (
    SELECT 1 FROM table
    WHERE table.id = col_in_table.id
))
```

6.17 Qualified Table Names and Aliases

- Columns can be prefaced by their relation, which in turn can be prefaced by the database name, using the syntax `database.relation.column`
- The * symbol can be qualified as well
- The `AS` keyword allows for column aliases with the syntax `SELECT database.column FROM table AS alias`
    - ↳ The alias name is enclosed in double quotes (although these are optional if the alias contains no spaces)
- Implicit aliases are not recommended but possible when the alias name contains no spaces, with syntax `SELECT database.column FROM table alias`
- Qualified table names are necessary when referring to two columns in different tables of the same name (for example, in a subquery, join, or union)
- Aliases are necessary when, for example, doing a self join

6.18 SQL Set Operations

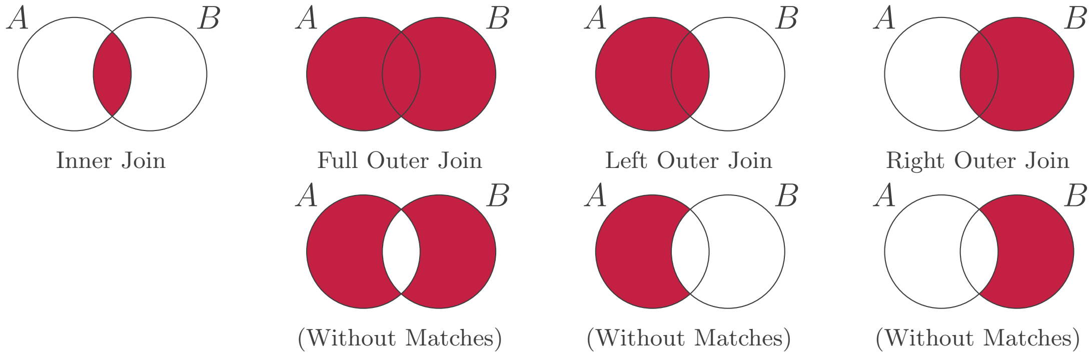
- Rows/values from two separate queries can be combined using `UNION`

```
SELECT Student.Name FROM Student
UNION
SELECT Staff.Name FROM Staff;
```

  - ↳ The columns need not have the same name; the only requirement is that the datatypes are the same (called union compatible)
    - ↳ In DB2, this includes the length of character attributes
    - ↳ In DB2, if an alias is specified for one table, the same alias must be specified for the other
  - ↳ `UNION` removes duplicates on the results; to avoid this, use `UNION ALL`
  - ↳ `UNION ALL` is more efficient, so it is desirable to use `UNION ALL` when possible (and perhaps let the processing program handle duplicates)
  - ↳ A technique used to distinguish which of the two tables a row originated from is to include a literal in the `SELECT` statement to serve as an indicator

- Set minus can be performed using `EXCEPT`
  - ↳ Similar to before, the two tables must be union compatible
  - ↳ `EXCEPT ALL` does not check for duplicates; this means that if the first table has more instances of a particular tuple than the second, then it will remain in the output
  - ↳ Thus, to use `EXCEPT ALL`, it is often necessary to ensure that the first table has no duplicates (such as by using an `EXISTS` subquery rather than a `JOIN`)
- `INTERSECT` and `INTERSECT ALL` perform set intersection

6.19 SQL Joins



Inner Join

Full Outer Join

Left Outer Join

Right Outer Join

(Without Matches)

(Without Matches)

(Without Matches)

- Joins are used to merge columns of tables, and the different types of joins describe how to handle mismatches
- The syntax is generally `SELECT * FROM table1 JOIN table2`, where `JOIN` can be any of the joints described
- `CROSS JOIN` performs a Cartesian product of the rows, forming every possible combination
  - ↳ This can be implicitly performed with `SELECT * FROM table1, table2`
- `INNER JOIN` is like `CROSS JOIN` but only combinations matching a predicate are returned, with syntax `INNER JOIN table2 ON predicate`
  - ↳ Often, the predicate (also called correlation condition) is checking equality of two columns, and there will not be multiple matches per row
  - ↳ This could also be performed with an implicit cross join follows by a filter with `WHERE`
- `FULL OUTER JOIN` include all rows matching a predicate like `INNER JOIN`, but rows without a match are also included with `NULL` values inserted

  - ↳ This is not implemented in SQLite
  - ↳ Instead, we can combine a left outer join and a right outer join without matches:

```
SELECT * FROM tableA
LEFT OUTER JOIN tableB
ON tableA.col = tableB.col
UNION ALL
SELECT * FROM tableB
LEFT OUTER JOIN tableA
ON tableB.col = tableA.col
WHERE tableA.col IS NULL;
```

- `LEFT OUTER JOIN` (or `LEFT JOIN`) includes all rows on the left, but not necessarily all rows on the right
- A left outer join without matches is performed by doing a left outer join then filtering on when the matched data is `NULL`
- Similar definitions for `RIGHT OUTER JOIN` and right outer join without matches are defined similarly
  - ↳ Right joins are not implemented in SQLite and are generally discouraged
- A self join merges a table with itself, based on two columns; this can be completed using aliases:

```
SELECT * FROM table AS t1
INNER JOIN table as t2
ON t1.column1 = t2.column2
AND t1.column2 = t2.column1
WHERE t1.column1 < t2.column1;
```

  - ↳ Two conditions are used to ensure both ways match (recall that `INNER JOIN` is just a refinement of `CROSS JOIN`, so we must check that both directions match)
  - ↳ The last filter removes duplicates
- A table is a preserved table in a join if all of its rows will be included; for example, the left table is preserved in either a left join or a full outer join
- Nulls in the original table and nulls in a non-preserved table are indistinguishable. If some form of filtering must be applied where this is a concern, then the filtering should occur in the `ON` clause, rather than the `WHERE` clause.

6.20 SQL Subqueries

- A subquery is a nested SQL query returning information (either individual values or a list of records) to an outer query
  - ↳ Subqueries are enclosed in parentheses
  - ↳ A subquery can either be non-correlated (could be run as an independent query) or correlated (references attributes in the outer query)
  - ↳ Subqueries can appear in many locations, including `SELECT` and `WHERE` clauses
- Subqueries can be used to test for set membership, make set comparisons, and determine set cardinality
- The `IN` keyword can be used with subqueries instead of lists:

```sql
-- Select artists who have not released an album
SELECT Artist.Name FROM Artist
WHERE Artist.Name NOT IN
(SELECT Album.ArtistId FROM Album);
```

- Another example is for finding entries which occur several times in the table, each time meeting a different condition:

```sql
-- Select courses offered in both fall and spring
SELECT name FROM courses
WHERE semester = 'Fall'
AND name IN
(SELECT name FROM courses
WHERE semester = 'Spring');
```

  ↳ Simply using `WHERE semester = 'Fall' AND semester = 'SPRING'` would yield no results

- `EXISTS` returns true if the subsequent subquery returns any results:

```sql
-- Select suppliers who make clothing
SELECT SupplierName FROM Suppliers
WHERE EXISTS (
    SELECT ProductName FROM Products
    WHERE Products.SupplierID = Supplier.SupplierID
    AND Category = 'Clothing'
);
```

  ↳ Note that the column returned by the subquery is irrelevant; it could be `*` or even `1`

  ↳ It is much more efficient to use `EXISTS` with a correlated subquery than `IN` with a non-correlated subquery, since it can terminate the instant a single row is identified (rather than having to evaluate the entire subquery). For example, to identify customers who have placed orders, the following two queries could be used, but the one using `EXISTS` is likely to be more efficient:

```sql
SELECT * FROM Customers
WHERE id IN (
    SELECT customer_id FROM Orders
);
```

```sql
SELECT * FROM Customers
WHERE EXISTS (
    SELECT 1 FROM Orders
    WHERE Customers.id = Orders.customer_id
);
```

- `ANY` and `ALL` allow a comparison between a field and a range of other values from a subquery, with syntax `WHERE column operator ANY|ALL (SELECT other_column FROM table WHERE condition)`

```sql
-- Select customers who have paid for a track of over 5 dollars
SELECT CustomerId FROM Invoice
WHERE InvoiceId = ANY(
    SELECT InvoiceId FROM InvoiceLine
    WHERE UnitPrice>5
);
```

↳ SQLite doesn't implement these operators, but `IN` can be used instead of `ANY`, and `min` and `max` can often be used to replicate the behavior of `ALL`

6.21 Group By

- `GROUP BY` groups records into summary rows, where one record is kept for each group
- `GROUP BY` is thus often used with aggregators like `count`, `sum`, etc.

```sql
-- Count how many tracks of each genre are present
SELECT GenreId, count(*)
FROM Track
GROUP BY GenreId;
```

- `GROUP BY` can work on multiple columns
- `HAVING` is used to filter `GROUP BY` results, similar to how `WHERE` is used to filter `SELECT` results
- Generally, it is best to use `WHERE` clauses when possible to do filtering ahead of time, then use `HAVING` when the condition depends on the value of aggregate functions

```sql
-- Count how many tracks of each genre are present, but only if they have at least 10 tracks
SELECT GenreId, count()
FROM Track
GROUP BY GenreId
HAVING count()>10;
```

- All columns appearing in the select statement must also appear in the `GROUP BY` clause; otherwise, it would be ambiguous which value should be returned for each group

6.22 SQLite Dates and Times

- SQL does not have a built-in datatype for dates or times, so often ISO 8601 is used to represent them with `TEXT` as `YYYY-MM-DD HH:MM:SS.SSS`
  - ↳ Hours range from 0 to 23, so there is no AM or PM
- With this format, lexicographic sort is chronological sort
- SQLite offers the functions `date`, `time`, and `datetime` which take an input, apply any modifiers, and return the resulting string in ISO 8601
- The valid inputs are:
  - ↳ Dates must be represented as `YYYY-MM-DD`, with all digits given and the correct delimiter
  - ↳ Times are `HH:MM`, `HH:MM:SS`, or `HH:MM:SS.SSS`
  - ↳ A time is optionally followed by a time-zone modifier `HH:SS` preceded by `+` or `-` (note that the opposite operation is used to calculate the time), or case-insensitive `Z` which is default time zone; whitespace may precede this

- ↳ Either a date, a time, or both (separated by whitespace, capital `T`, or both) are valid inputs
- ↳ `now`, case-insensitive, selects the current time
- ↳ It can also accept a Julian day number with arbitrary precision as an integer or floating point

- Any number of case-insensitive modifiers can then be passed to the function as separate strings, separated by commas, including the following:
  - ↳ A (possibly signed) number/decimal followed by `years`, `months`, `days`, `hours`, `minutes`, or `seconds`, where the trailing `s` is optional
  - ↳ `start of month`, `start of year`, and `start of day` shift the date/time back to the start
  - ↳ `weekday N` shifts the date forward to the next date where the weekday number is `N`, where Sunday is 0 and Saturday is 6
- After converting to date-time format, operations like `MIN` and `MAX` work appropriately
- In general, however, it is better to use non-SQL tools for handling dates and times; one option is the `datetime` module:
  - ↳ The `date` class has a constructor accepting year, month, then day, or `date.today()`
  - ↳ The `time` class has a constructor accepting optional hour, minute, second, and microsecond
  - ↳ The `datetime` class has a constructor accepting date fields and then optional time fields, or `datetime.now()`
  - ↳ The `timedelta` class represents a difference in times/dates with optional arguments days, seconds, microseconds, milliseconds, minutes, hours, and weeks; this can be added/subtracted from other datetime objects
- Datetime objects can be converted to ISO 8601 with `obj.strftime("%Y-%m-%d %H:%M:%S.%f%z")`, and the reverse operation can be done with `obj=datetime.strptime(dt_string, "%Y-%m-%d %H:%M:%S.%f%z")`
- sqlite3 has built-in support for Python datetime objects through `DATE` and `TIMESTAMP` types
  - ↳ To use these, specify `detect_types = sqlite3.PARSE_DECLTYPES | sqlite3.PARSE_COLNAMES` when calling the `connect` function

6.23 DB2 Dates and Times

- Information about the current date and time can be obtained with the literals `CURRENT_TIMESTAMP`, `CURRENT_DATE`, and `CURRENT_TIME`
- A duration literal can be specified by providing a number followed by one of `YEARS`, `MONTHS`, `DAYS`, `HOURS`, `MINUTES`, `SECONDS`, and `MICROSECONDS`
  - ↳ This can then be added to or subtracted from a date, time, or timestamp datatype
  - ↳ When adding, first add `YEARS`, then `MONTHS`, and so on
  - ↳ When subtracting, first subtract `MICROSECONDS`, then `SECONDS`, and so on
  - ↳ Two dates can be subtracted to yield a duration `yyyymmdd`, two times can be subtracted to yield a duration `hhmmss`, and two timestamps can be subtracted to yield a duration `yyyymmddhhmmss.mmmmmm`
  - ↳ Integer quantities can be extracted from durations/dates/times with scalar functions `YEAR`, `MONTH`, `DAY`, `HOUR`, `MINUTE`, `SECOND`, and `MICROSECOND`
  - ↳ Character strings can be converted to date/time types with the scalar functions `DATE` and `TIME`

6.24 Stored Procedures

- A stored procedure is a function written in SQL and stored in the database
  - ↳ SQLite does not offer these, since the client can directly interact with the database (and can use functions they define in Python or whatever other language is being used)
  - ↳ The following syntax and discussion is for MySQL
- An example of a stored procedure is:

```
DELIMITER //
CREATE PROCEDURE add_now_to_log()
    BEGIN
    INSERT INTO log VALUES (datetime('now'));
    END //
DELIMITER ;
```

  - ↳ The stored procedure is then called with `CALL add_now_to_log();`
  - ↳ Since `;` is used in the stored procedure, the delimiter must be switched to something else while defining the procedure, so that the full procedure is read before it is created; common choices are `//` and `$$`
- Variables in a procedure are declared with `DECLARE var_name data_type DEFAULT value`, where the default value is optional
  - ↳ The variable is then initialized/updated with `SET var_name = value;` or `SELECT expr INTO var_name FROM table;`
  - ↳ Session variables are user-defined variables outside of procedures that last for the scope of the connection; they are preceded by `@` and do not require declaration
- Stored procedures can accept one of three types of parameters:

  (i) `IN`, the default mode, which must be set by the caller and cannot be modified by the procedure

  (ii) `OUT`, which the procedure is allowed to set but cannot read

  (iii) `INOUT`, which allows the procedure to read and write to it

```
DELIMITER //
CREATE PROCEDURE increment (INOUT tally INT(4), IN inc INT(4))
    BEGIN
    SET tally = tally + inc;
    END //
DELIMITER ;
SET @count = 0;
CALL increment(@count, 3);
SELECT @count;
```

- A stored procedure can use a `CASE` statement to execute different statements, such as:

```
CASE
  WHEN cond_1 THEN statement_1;
  WHEN cond_2 THEN statement_2;
  ELSE statement_3;
END CASE;
```

- Some advantages of stored procedures include:

  - Performance: stored procedures are compiled and stored in the database, allowing for caching and faster performance on repeated calls
  - Less traffic: lengthy SQL queries need not be sent over the connection
  - Reusable: Stored procedures can be used by different applications and users
  - Secure: Users can be given access to the stored procedures rather than the full tables

- Some disadvantages of stored procedures include:
  - Memory and CPU usage: Stored procedures add overhead to each connection
  - Difficult to debug: Most database management systems do not have good support for traceback and error reporting
  - Difficult to maintain: Stored procedures are less transparent and harder to write, especially if the schema changes

6.25 Triggers

- A trigger automatically performs some pre-defined set of queries when a particular action is carried out by the user
- The general syntax is:

```
CREATE TRIGGER trigger_name [timing] event_name
ON table_name [WHEN predicate] BEGIN
    commands;
END;
```

- The timing specifies when the trigger logic is applied
  - For tables it is either `BEFORE` (default) or `AFTER`
  - `BEFORE` is often used for validation, whereas `AFTER` is used for updating other tables
  - For a view, the timing should be `INSTEAD OF`, which is used to redirect the action (so that the view behaves like a table from the user-end)
- The event name is either `INSERT`, `DELETE`, or `UPDATE`
- Within the trigger logic, the pseudo table `NEW` refers to the row's new values (for `INSERT` and `UPDATE`) and `OLD` refers to the old values (for `UPDATE` and `DELETE`)
  - It can only be used to access particular values, like `NEW.col_name`; it cannot be treated like a table for the purposes of joins or `NEW.*`

```
CREATE TRIGGER id_change AFTER UPDATE ON students WHEN NEW.id != OLD.id
BEGIN
    UPDATE log SET student_id = NEW.id WHERE student_id = OLD.id;
END;
```

- To fire the trigger when only specific columns are updated, one can use `UPDATE OF col1, col2`
- To raise an error, use `RAISE(action, message)`
  - The main action used is `ABORT`, which ends the query and undoes any changes
- A trigger can be removed with `DROP TRIGGER trigger_name`, or if the trigger may not exist, `DROP TRIGGER IF EXISTS trigger_name`
- Some advantages of triggers include:
  - They always activate when the operation occurs

  - ↳ They are logic stored in the database, not the application/connection
  - ↳ They are useful for auditing and validation

- Some disadvantages of triggers include:
  - ↳ They are not transparent, and may activate without the user's knowledge
  - ↳ They can lead to inefficiencies, especially if they are running unknown
  - ↳ Stored procedures are often a better alternative

6.26 DDL Operations

- Altering or dropping a table can cause pre-existing constraints (such as foreign keys and checks) to be lost
- SQLite supports two actions with `ALTER TABLE`:
  - ↳ Renaming a table, with `ALTER TABLE old_name RENAME TO new_name;`
  - ↳ Adding a column to a table, with `ALTER TABLE table_name ADD COLUMN col_name type;`
- A table can be removed with `DROP TABLE table_name`, or if the table may not exist, `DROP TABLE IF EXISTS table_name`
  - ↳ This implicitly drops all rows from the table first

6.27 TCL Operations

- `COMMIT` and `ROLLBACK` are used to terminate transactions
- `SAVEPOINT savepoint-name` sets a savepoint which can be restored with `ROLLBACK TO SAVEPOINT savepoint-name`
  - ↳ In DB2, the savepoint name defaults to the last made savepoint when performing a rollback, but it must be specified in SQLite
- `RELEASE SAVEPOINT savepoint-name` removes a savepoint
  - ↳ This is automatically done on a commit
- Formally, when a savepoint is created, it initiates a new nested transaction, which is committed when the savepoint is released and rolled back when the savepoint is rolled back

6.28 Views

- A view is a named `SELECT` statement (a virtual table composed of the result of the query), defined with `CREATE VIEW view_name AS SELECT ...`
- A view has rows and columns, but instead of holding data, it points to data held in other tables (and changes when the referenced table changes)
- Normal select statements work on views, but views are read-only
- Views are useful for creating read-only tables, hiding data complexity (such as with joins), customizing data (using functions and group by), and for privacy
- Views can be slow, especially when creating views of other views
- A view is dropped automatically when the referenced table is dropped
  - ↳ To drop a view automatically when the connection closes, use `CREATE TEMPORARY VIEW`
  - ↳ A view can be manually dropped with `DROP VIEW` or `DROP VIEW IF EXISTS`

6.29 Indices

- See section 8 for a discussion of indices
- Most ordinary tables in SQLite contain an implicitly-created `rowid` column that is used to uniquely identify rows (for indexing the database)
  - ↳ This can be turned off with `WITHOUT ROWID` at the end of the `CREATE` statement
  - ↳ If an integer primary key is declared, then this becomes an alias for `rowid`
- Additional indices can be created with `CREATE INDEX index_name ON table (col);`
  - ↳ Multiple columns can also be included instead of just one
  - ↳ To remove an index, use `DROP INDEX index_name` or `DROP INDEX IF EXISTS index_name`

6.30 Select From Final Table (DB2)

- To select data from rows that were added/modified by a query, the DML query can be embedded inside the FROM clause of the following:

```
SELECT ...
  FROM FINAL TABLE (
    [dml query])
;
```

- This is useful either for confirming proper updates/deletes, or for obtaining the value of an automatically-filled attribute during inserts

6.31 Merge Statement (DB2)

- To either update rows that exist, or add rows if they do not, a `MERGE` statement can be used as follows:

```
MERGE INTO table-name
  USING (VALUES (lookup-column-values)) AS alias-name (column-names)
     ON matching-criteria
  WHEN MATCHED
    AND conditions
    THEN UPDATE SET col1 = val1,
                    col2 = val2,
                    ...
  WHEN NOT MATCHED
    THEN INSERT (columns-to-insert-into)
         VALUES (values-to-insert)
;
```

- An arbitrary select statement can be included in the `USING` clause
- Multiple `WHEN MATCHED` clauses can be provided with different conditions; only the first one satisfied is executed
- An example is:

```
MERGE INTO ED03DBB.ED03E4T0 C
  USING (SELECT ASSOC_ID
           FROM ED03DBB.ED03E1T0
          WHERE ASSOC_DEPT_ID = '600') AS A (ASSOC_ID)
    ON C.COMP_ASSOCIATE_ID = A.ASSOC_ID
  WHEN MATCHED
    THEN UPDATE SET COMP_MANUFACTURER = 'Dell',
                    COMP_TYPE = 'Laptop',
                    COMP_IN_SERVICE_DTE = CURRENT_DATE - 14 DAYS,
                    COMP_USER_ID = SESSION_USER,
                    COMP_ACTIVITY_TSP = CURRENT_TIMESTAMP
  WHEN NOT MATCHED
    THEN INSERT (COMP_COMPUTER_ID,
                 COMP_ASSOCIATE_ID,
                 COMP_MANUFACTURER,
                 COMP_TYPE,
                 COMP_IN_SERVICE_DTE,
                 COMP_USER_ID,
                 COMP_ACTIVITY_TSP)
         VALUES ('HP600' || STRIP(CHAR(ASSOC_ID)),
                 ASSOC_ID,
                 'HP',
                 'Desktop',
                 CURRENT_DATE - 14 DAYS,
                 SESSION_USER,
                 CURRENT_TIMESTAMP)
;
```

6.32 Miscellaneous SQL

- Parameterized queries help protect against SQL injection attacks by allowing Python objects to be directly passed to SQL statements
  - ↳ The sqlite3 module handles string conversion and ensures all characters are properly escaped
  - ↳ For example, `conn.execute("INSERT INTO students VALUES (?, ?);", name, age)`
  - ↳ A tuple containing the values of `name` and `age` could also be passed (i.e., they do not need to be unpacked)

7. Transaction Management

7.1 Introduction

- A transaction is a unit of program execution that accesses and possibly updates various data items
- The principles of transactions for relational databases are summarized with then acronym ACID (called the ACID Properties):
  - A: Atomicity. Despite consisting of possibly many statements, the collection must appear to the user as an indivisible unit: it either executes in entirety or not at all (if failure occurs for any reason, any changes must be undone)
  - C: Consistency. The database must remain in a consistent state after an isolated transaction is run (data still is valid)
  - I: Isolation. Since transactions consist of many statements, it must be ensured that each transaction is not affected by other concurrent transactions
  - D: Durability. Successful transactions must remain permanent even if the system later crashes
- The primary way to make a database system more efficient is to allow for concurrency, but the scheduling of transactions cannot violate the ACID test; it must appear to the end-user that each query runs independently
  - ↳ Note that no restrictions are placed on the order in which transactions are executed; if an order is required, they should be packaged in the same transaction
  - ↳ All transactions must conform to the "Correctness Principle": if a transaction executes independently without system errors, and the database begins in a consistent state, then it must end in a consistent state
  - ↳ When determining how to implement concurrency, there is a tradeoff between latency (wait-time for a given query) and throughput (the number of queries that can be processed at once)

7.2 Serializable Schedules

- A schedule is a serial schedule if the actions consist of all the actions from one transaction, then all the actions of another transaction, and so on
- A schedule is a serializable schedule if there exists a serial schedule producing the same output
  - ↳ Serializable schedules may have actions being interweaved between transactions, resulting in concurrency, but it appears to the end user as if each runs in isolation
  - ↳ A non-serializable schedule violates ACID properties, so may not be used by the database scheduler
- When determining whether a schedule is serializable, only the details of what element is being written to/read from is used, not the actual contents of the read/write
  - ↳ This means arithmetic coincidence cannot be utilized; for example, properties such as commutativity and associativity cannot be utilized
  - ↳ The reason for this is understanding the effect of a query on the database is a hard problem, related to the Halting Problem

7.3 Conflict-Serializable Schedules

- A conflict is a pair of consecutive actions in a schedule such that the output may change if their order is interchanged
  - ↳ To identify conflicts (again assuming no arithmetic coincidence), we notate the operations as $r_i(X)$ and $w_i(X)$ for read and write, where i is the transaction and X is the database element
  - ↳ Note that a write represents a blind write, where the element is written to without being read, and is instead being overwritten

- Consider two actions $a_i(X)$ and $b_j(Y)$. We have:
  (i) There is a conflict if $i = j$, since we never allow actions within the same transaction to be re-ordered
    - ↳ Although a transaction is represented by r_i and w_i, other database-independent actions are also being performed; for example, the information of a read can be processed in some way and then used in a write
    - ↳ Thus, it is not feasible to consider re-ordering steps within a transaction under this framework

  (ii) Otherwise, if $X \neq Y$, there is no conflict, since different database elements are being utilized

  (iii) Finally, if $X = Y$, then there is a conflict iff either a or b is a write action

- Two schedules are conflict-equivalent if one can be transformed to the other via non-conflicting swaps of adjacent actions

- A schedule is conflict-serializable if it is conflict equivalent to a serializable schedule
  - ↳ Every conflict-serializable schedule is serializable
  - ↳ Not every serializable schedule is conflict-serializable, such as $r_1(X); w_2(X); w_1(X); w_3(X)$. This is not conflict serializable since $w_2(X)$ and $w_1(X)$ conflict, but it is serializable since in any order, the result is over-written by $w_3(X)$
  - ↳ In practice, it is difficult to determine whether a schedule is serializable, but it is much easier to determine whether it is conflict-serializable

- Transaction T_1 takes precedence over transaction T_2 in schedule S if there are actions $A_1 \in T_1$ and $A_2 \in T_2$ such that A_1 precedes A_2 in S, A_1 and A_2 involve the same data element, and at least one is a write
  - Equivalently, T_1 has precedence over T_2 if T_1 occurs before T_2 and there exists conflicting actions $A_1 \in T_1$ and $A_2 \in T_2$
  - This is denoted $T_1 < T_2$

- In a conflict-serializable schedule, $T_i < T_j$ must imply that T_i precedes T_j in the corresponding serial schedule
  - ↳ Thus, by building a precedence graph (a directed graph with edges from T_i to T_j for each $T_i < T_j$), a schedule is conflict-serializable iff it is acyclic
  - ↳ A conflict-equivalent serial schedule can then be found via topological sorting

7.4 Locks

- Locks provide a mechanism for producing conflict-serializable schedules while still allowing concurrency; transactions request and release locks on different elements of the database

- Locks control transactions as follows:
  - ↳ A transaction may only read or write to an element if it was previously granted a lock and has not released the lock yet
  - ↳ Transactions must eventually release all locks
  - ↳ No two transactions can concurrently have a lock on the same element

- Locking an element is denoted $l_i(X)$, and releasing a lock is denoted $u_i(X)$

- Simply having locks is not enough to ensure conflict-serializable schedules, since each action could simply lock and then immediately unlock the element

- Two-Phase Locking (2PL) guarantees that a legal schedule of consistent transactions is conflict-serializable by forcing all lock actions to precede all unlock actions
  - ↳ The expanding phase consists of the portion of the transaction where locks may be acquired, and the shrinking phase consists of the portion where locks may be released

- ↳ Intuitively, each transaction can be viewed as executing in entirety the moment it releases its first lock (and this is what the corresponding serial schedule looks like)

- To produce the schedule, each transaction can greedily request resources/locks, and the scheduler will either grant these are deny them

7.5 Types of Locks

- If multiple transactions only read from an element without writing to it, then hypothetically these could run concurrently in any order
- To allow this, locks are divided into two categories: shared locks and exclusive locks
- A shared lock is used for reading, while an exclusive lock is used for reading and/or writing
- Many shared locks can be granted simultaneously, but an exclusive lock can only be granted if it is the only lock
- Shared lock requests are denoted $sl_i(X)$ and exclusive locks are denoted $xl_i(X)$
- Thus, all of the rules become:
  - ↳ Consistency of Transactions: In any transaction T_i, $r_i(X)$ must be preceded by $sl_i(X)$ or $xl_i(X)$, with no intervening $u_i(X)$; $w_i(X)$ must be preceded by $xl_i(X)$ with no intervening $u_i(X)$; and $sl_i(X)$ and $xl_i(X)$ must eventually be followed by $u_i(X)$
  - ↳ Two-Phase Locking of Transactions: No action $sl_i(X)$ or $xl_i(X)$ may be preceded by $u_i(Y)$ for any Y
  - ↳ Legality of Schedules: If $xl_i(X)$ appears in a schedule, then no $xl_j(X)$ or $sl_j(X)$ may appear for $j \neq i$ until $u_i(X)$ appears; and if $sl_i(X)$ appears in a schedule, then there cannot be a following $xl_j(X)$ for $j \neq i$ until $u_i(X)$ appears

7.6 Transactions in SQLite

- SQLite chooses to lock on the entire database at a time, rather than locking on a table/row/value
  - ↳ This sacrifices efficiency for ease of implementation
  - ↳ Due to triggers, it may be difficult to determine how a given action impacts the database
  - ↳ This is unlike DB2, which locks by rows using SHARED locks, UPDATE locks (one of which can be granted to allow for updates while still allowing shared locks), and EXCLUSIVE locks
- There are five locking states of the database:
  - ↳ UNLOCKED. No locks are held on the database, and the database cannot be read or written. This is the default state.
  - ↳ SHARED. The database may be read but not written. Any number of processes can hold a shared lock, so many simultaneous reads can occur.
  - ↳ RESERVED. A single process has a reserved lock, indicating that it plans to eventually write to the database, but that shared locks can continue being acquired until then.
  - ↳ PENDING. A single process has a pending lock, indicating that it plans to write to the database as soon as possible. No new shared locks may be acquired, although existing shared locks are allowed to finish.
  - ↳ EXCLUSIVE. A single process may have an exclusive lock, and no other locks may be present at that time. This lock permits reads and writes. Since this limits concurrency, SQLite minimizes the number of these locks granted.
- A transaction in SQLite ends with either a commit, rollback, or error (which in turn triggers a rollback)
  - ↳ These are denoted in a schedule with commit_i and rollback_i

  - ↳ Typically, the transaction does not write to the database until the commit, and instead modifies a local cached copy (so that different transactions may see different snapshots of the database)
  - ↳ More formally, SQLite uses write-ahead logging (WAL) to record changes to the database (so a literal copy is not made, but this is an easy way to think about it)
  - ↳ In the case of a rollback/error, all changes are undone
  - ↳ All locks are released upon commit

- A transaction can be run in three modes:
  - ↳ DEFERRED. A shared lock is requested upon the first read, and an exclusive lock is requested upon the first write (recall that the write typically occurs on commit, not on the first update/insert statement). This is the default.
  - ↳ IMMEDIATE. A reserved lock is requested immediately, and it will fail or wait if another transaction is holding an exclusive/reserved/pending lock. An exclusive lock is requested prior to writing (typically the promotion occurs on commit).
  - ↳ EXCLUSIVE. An exclusive lock is requested immediately, and it will fail or wait if other locks are being held.

- The syntax in SQLite is BEGIN [MODE] TRANSACTION for beginning the transaction (where [MODE] is optionally passed), and either COMMIT TRANSACTION or ROLLBACK TRANSACTION.

- Statements executed outside of explicit transactions are handled differently depending on the isolation_level property specified when forming the connection
  - ↳ `'DEFERRED'`, `'IMMEDIATE'`, or `'EXCLUSIVE'`. Transactions in these modes are implicitly opened in the respective mode before any INSERT, UPDATE, DELETE, or REPLACE statement is run. The transaction must be explicitly committed or rolled back with `commit()` or `rollback()`, otherwise the changes are rolled back. The default is `'DEFERRED'`.
  - ↳ `None`. Each executed statement is treated as its own transactions, automatically committed after it finishes. This mode is often better to use, since it is better to explicitly open transactions when they are needed.

- If a statement cannot be run because it cannot acquire the locks, then SQLite issues an error after a certain amount of time

7.7 Rollbacks

- A rollback aborts a transaction and undoes its changes (which is necessary by atomicity)

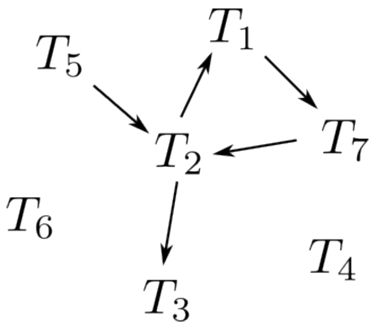
- If a serial schedule is not used, rolling back one transaction may require that previous transactions be rolled back as well
  - ↳ Namely, if a transaction T_1 has read modified (uncommitted) data created by T_2, then a rollback of T_2 also requires that T_1 be rolled back
  - ↳ This can lead to cascading rollbacks
  - ↳ Cascading rollbacks can be avoided using strict two-phase locking, which only allows locks to be released at the moment of commit or rollback, and not sooner
  - ↳ There is a tradeoff between throughput and preventing cascading rollbacks

7.8 Deadlock

- In a locking schedule, deadlock occurs when transactions are waiting for locks to be released, but they are holding the locks that other transactions need to continue

- Deadlock can be detected via a Wait-For graph, where each transaction is a node, and directed edges point from the waiting transaction to the transaction it is waiting on
  - ↳ If the Wait-For graph has a cycle, then there is deadlock

- ↳ Deadlock must be resolved by rolling back a transaction
- ↳ However, many methods do not directly detect deadlock via waits-for graphs, and instead impose other rules



All are deadlocked except T_3, T_4, and T_6

- Using a timeout is one method of deadlock resolution, where any transaction waiting for some number of seconds is rolled back
  - ↳ Choosing the threshold is difficult and may not solve the issue if the transactions are resubmitted the same way
- The Wait-Die and Wound-Wait methods resolve deadlocks by rolling back transactions based on their timestamps when one transactions requests the resources/locks of another
  - ↳ No explicit deadlock detection is performed; rather, the rules make it so that deadlock could not possibly occur
  - ↳ Each transaction T_i is assigned a timestamp $ts(T_i)$ for when it starts, where $ts(T_i) < ts(T_j)$ indicates that T_i is older (such as with Unix time)
  - ↳ Both methods give priority to older transactions, but in different ways
  - ↳ When a transaction is rolled back, then it restarts with the same timestamp, so that it will eventually be the oldest (being rolled back does not penalize its priority)
  - ↳ A waiting transaction tries to get the lock again as soon as it is free, but when a rolled back transaction resurfaces is less predictable
- When T_i requests a lock held by T_j, the Wait-Die deadlock resolution method performs one of two actions:
  - ↳ If $ts(T_i) < ts(T_j)$, T_i waits for T_j (older waits for younger)
  - ↳ Otherwise, T_i aborts (younger dies)
- Similarly, the Wound-Wait deadlock resolution method takes the actions:
  - ↳ If $ts(T_i) < ts(T_j)$, then T_j aborts (older wounds younger, forcing it to roll back)
  - ↳ Otherwise, T_i waits (younger waits for older)
- Both these methods minimize starvation, which is when objects are less efficient due to waiting for resources (deadlock is the most extreme form of this)
  - ↳ Wait-Die tends to roll back more transactions than Wound-Wait (since all younger transactions die when requesting resources from older), but the transactions have done less work when they are rolled back
- These methods do not specify when each transaction is allocated time to run its actions; one method is round-robin style scheduling, which performs one action from each transaction in an alternating fashion
  - ↳ Commit is treated as its own action
  - ↳ If a transaction is waiting on a lock, it can try again next time (but this means that younger transactions may acquire it sooner)

7.9 Validation Scheduling

- Locks are a form of pessimistic concurrency control because they assume transactions will conflict unless the scheduler prevents them
- Validation is an optimistic way of ensuring serializable schedules, which assumes that conflicts are rare
  - ↳ This is useful when most users are readers, and only small fractions of the total database are accessed/modified by any transaction
- Each transaction has three phases:
  - ↳ A read phase, where all writes occur to private storage
  - ↳ A validation phase, where it is checked that no conflicts will occur if the write phase happens
  - ↳ A write phase, where the writes are made public if the validation is successful (otherwise the transaction is aborted)
- In particular, validation ensures that the schedule generalized will be equivalent to the serial schedule ordered by timestamp (all conflicts must occur in one direction)
- The validation test for T_j check that for all $ts(T_i) < ts(T_j)$, we have one of the following:
  1. T_i completes its write phase before T_j starts its read phase
     - ↳ This is what would occur in a serial schedule
  2. (T_i does not write to any object that T_j reads from) and (T_i completes its write phase before T_j starts its write phase)
     - ↳ Writes/reads from T_i can be interweaved with reads of T_j as long as there is no WR conflict
     - ↳ No writes are interweaved
  3. (T_i does not write to any object that T_j reads from) and (T_i does not write to the same element that T_j writes to) and (T_i completes its read phase before T_j completes its read phase)
     - ↳ Writes may now be interweaved as well, under the condition that writes in T_i act on objects which T_j does no interact with
     - ↳ To ensure T_j (the later transaction) does not impact T_i (the earlier transaction), we must have that T_i completes its read phase before T_j completes its write phase (i.e., T_j cannot begin to write while T_i is still reading, so that all conflicts are resolved as RW)

7.10 Timestamp-Based Scheduling

- Timestamp-based scheduling is also optimistic, but validates each action as it occurs, rather than all at once in a validation phase
- To accomplish this, every database element has the additional metadata:
  - ↳ $RT(X)$: The read time of X, which is the most recent time X was read
  - ↳ $WT(X)$: The write time of X, which is the most recent time X was written to
  - ↳ $C(X)$: The commit bit of X, which is a boolean for whether the most recent write to X has been committed
- Physically unrealizable behaviors are those that yield results incapable of being produced by the serial schedule where each transaction runs in order according to its start time
  - ↳ This method aims to prevent these behaviors
  - ↳ For T_i beginning before T_j, read-too-late occurs when $r_i(X)$ occurs after $w_j(X)$, and write-too-late occurs when $w_i(X)$ occurs after $r_j(X)$
  - ↳ These are essentially the conflicts from before, aside from write-write conflicts - in this case, T_i can just skip its write action by the following rule

- The Thomas Write Rule states that write may be skipped if a later write is in place
  - ↳ The one exception is when the later write is aborted (in which case the earlier write should have occurred)
  - ↳ In this case, these writes are "delayed" until either the later transaction is committed (in which case they do not run) or aborted (in which case they do run). This is kept track of with the committed bit
- The rules for a transaction T_i doing a read $r_i(X)$ are:
  - (a) If $ts(T_i) > WT(X)$, then the read is physically realizable, so:
    - (i) If $C(X)$ is true, then perform $r_i(X)$ and update $RT(X) = \max\{ts(T_1), RT(X)\}$
    - (ii) If $C(X)$ is false, then delay $r_i(X)$ until the transaction which wrote to X commits or aborts
  - (b) If $ts(T_i) < WT(X)$, then the read is physically unrealizable (read-too-late), so roll back and restart with a new timestamp
- The rules for a transaction T_i doing a write $w_i(X)$ are:
  - (a) If $ts(T_i) > RT(X)$ and $ts(T_i) > WT(X)$, then it is physically realizable, so store a copy of X in case of rollback, perform $w_i(X)$, and set $WT(X) = ts(T_i)$ and $C(X)$ to false
  - (b) If $ts(T_i) > RT(X)$ and $ts(T_i) < WT(X)$, then it is physically realizable but has already been written to, so:
    - (i) If $C(X)$ is true, then the Thomas Write Rule applies, so do nothing
    - (ii) If $C(X)$ is false, then delay the action until the other transactions commits or aborts
  - (c) If $ts(T_i) < RT(X)$, then the write is physically unrealizable (write-too-late), so roll back and restart with a new timestamp
- When T_i commits, it finds all of the elements it wrote to and sets the commit bits to true
- When T_i rolls back, restore the old values of any elements written to, and allow the waiting transactions to perform reads/writes

7.11 Comparison of Concurrency Control Methods

- For the optimistic methods, validation and timestamp-based methods have similar performance and rates of rollback
  - Validation has overhead all at once during the validation phase, while timestamp methods have overhead distributed across all actions
- Storage requirements for locking and validation methods are similar
  - ↳ Locking has storage requirements proportional to the number of elements locked
  - ↳ Validation needs to store timestamps and the sets of elements accessed/modified for all open transactions
- Locking has few rollbacks, while validation has more frequent rollbacks
- Validation has fewer delays in transactions, while locking can cause delays due to starvation

8. Indices

8.1 Introduction

- Indices can improve the speed of many operations:
    - ↳ Looking up rows in a way not according to their primary key
    - ↳ Identifying matching rows according to the value in a column, when doing a look-up
- However, indices add additional time to CRUD operations (since indices must be updated) and take more memory
    - ↳ Indices for attributes that change often should be reduced
    - ↳ The 80-20 rule says that 80% of the database's processing time is caused by 20% of the queries, so identifying and optimizing for the 20% is most important
    - ↳ Some queries require faster response times than others based on context, so this can be used when planning which indices to create and which to avoid
- A primary index is an index which specifies the order of the stored data, while a secondary index provides alternative means of accessing data in the relation
- Indices can be unsorted (such as hash tables, which only allow records to be found through equality) or sorted (which allow records to be returned matching a range)
    - ↳ Sorted indices are most common and generally most useful
- Primary indices can be dense (containing an entry for every search key value) or sparse (containing only some entries for the search key)
    - ↳ For example, an index containing the first word of each starting letter in a dictionary is a sparse index
    - ↳ Secondary indices cannot be sparse since the index does not correspond to the order in which the data is stored
    - ↳ Most indices are dense

Query Planning

Overview

The best feature of SQL (in all its implementations, not just SQLite) is that it is a *declarative* language, not a *procedural* language. When programming in SQL you tell the system *what* you want to compute, not *how* to compute it. The task of figuring out the *how* is delegated to the *query planner* subsystem within the SQL database engine.

For any given SQL statement, there might be hundreds or thousands or even millions of different algorithms of performing the operation. All of these algorithms will get the correct answer, though some will run faster than others. The query planner is an AI that tries to pick the fastest and most efficient algorithm for each SQL statement.

Most of the time, the query planner in SQLite does a good job. However, the query planner needs indices to work with. These indices must normally be added by programmers. Rarely, the query planner AI will make a suboptimal algorithm choice. In those cases, programmers may want to provide additional hints to help the query planner do a better job.

This document provides background information about how the SQLite query planner and query engine work. Programmers can use this information to help create better indexes, and provide hints to help the query planner when needed.

Additional information is provided in the SQLite query planner and next generation query planner documents.

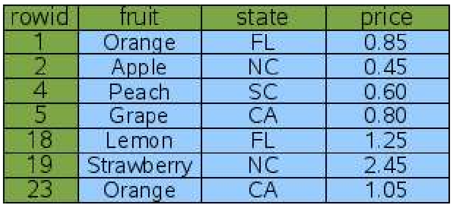
1. Searching

1.1. Tables Without Indices

Most tables in SQLite consist of zero or more rows with a unique integer key (the rowid or INTEGER PRIMARY KEY) followed by content. (The exception is WITHOUT ROWID tables.) The rows are logically stored in order of increasing rowid. As an example, this article uses a table named "FruitsForSale" which relates various fruits to the state where they are grown and their unit price at market. The schema is this:

```
CREATE TABLE FruitsForSale(
  Fruit TEXT,
  State TEXT,
  Price REAL
);
```

With some (arbitrary) data, such a table might be logically stored on disk as shown in figure 1:



rowid	fruit	state	price
1	Orange	FL	0.85
2	Apple	NC	0.45
4	Peach	SC	0.60
5	Grape	CA	0.80
18	Lemon	FL	1.25
19	Strawberry	NC	2.45
23	Orange	CA	1.05

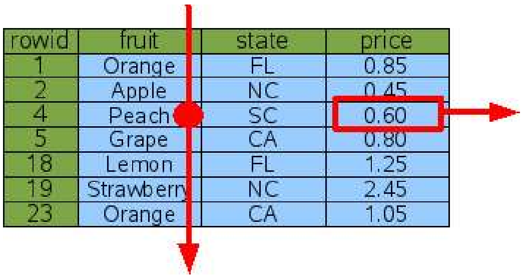
Figure 1: Logical Layout Of Table "FruitsForSale"

In this example, the rowids are not consecutive but they are ordered. SQLite usually creates rowids beginning with one and increasing by one with each added row. But if rows are deleted, gaps can appear in the sequence. And the application can control the rowid assigned if desired, so that rows are not necessarily inserted at the bottom. But regardless of what happens, the rowids are always unique and in strictly ascending order.

Suppose you want to look up the price of peaches. The query would be as follows:

```
SELECT price FROM fruitsforsale WHERE fruit='Peach';
```

To satisfy this query, SQLite reads every row out of the table, checks to see if the "fruit" column has the value of "Peach" and if so, outputs the "price" column from that row. The process is illustrated by figure 2 below. This is algorithm is called a *full table scan* since the entire content of the table must be read and examined in order to find the one row of interest. With a table of only 7 rows, a full table scan is acceptable, but if the table contained 7 million rows, a full table scan might read megabytes of content in order to find a single 8-byte number. For that reason, one normally tries to avoid full table scans.



rowid	fruit	state	price
1	Orange	FL	0.85
2	Apple	NC	0.45
4	Peach	SC	0.60
5	Grape	CA	0.80
18	Lemon	FL	1.25
19	Strawberry	NC	2.45
23	Orange	CA	1.05

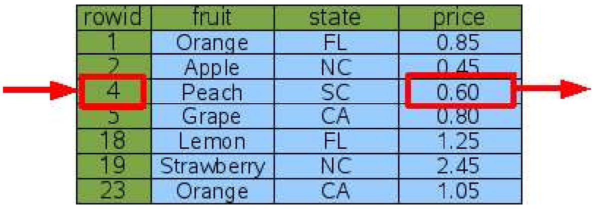
Figure 2: Full Table Scan

1.2. Lookup By Rowid

One technique for avoiding a full table scan is to do lookups by rowid (or by the equivalent INTEGER PRIMARY KEY). To lookup the price of peaches, one would query for the entry with a rowid of 4:

```
SELECT price FROM fruitsforsale WHERE rowid=4;
```

Since the information is stored in the table in rowid order, SQLite can find the correct row using a binary search. If the table contains N elements, the time required to look up the desired row is proportional to logN rather than being proportional to N as in a full table scan. If the table contains 10 million elements, that means the query will be on the order of N/logN or about 1 million times faster.



rowid	fruit	state	price
1	Orange	FL	0.85
2	Apple	NC	0.45
4	Peach	SC	0.60
5	Grape	CA	0.80
18	Lemon	FL	1.25
19	Strawberry	NC	2.45
23	Orange	CA	1.05

Figure 3: Lookup By Rowid

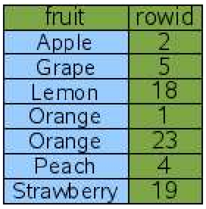
1.3. Lookup By Index

The problem with looking up information by rowid is that you probably do not care what the price of "item 4" is - you want to know the price of peaches. And so a rowid lookup is not helpful.

To make the original query more efficient, we can add an index on the "fruit" column of the "fruitsforsale" table like this:

```
CREATE INDEX Idx1 ON fruitsforsale(fruit);
```

An index is another table similar to the original "fruitsforsale" table but with the content (the fruit column in this case) stored in front of the rowid and with all rows in content order. Figure 4 gives a logical view of the Idx1 index. The "fruit" column is the primary key used to order the elements of the table and the "rowid" is the secondary key used to break the tie when two or more rows have the same "fruit". In the example, the rowid has to be used as a tie-breaker for the "Orange" rows. Notice that since the rowid is always unique over all elements of the original table, the composite key of "fruit" followed by "rowid" will be unique over all elements of the index.



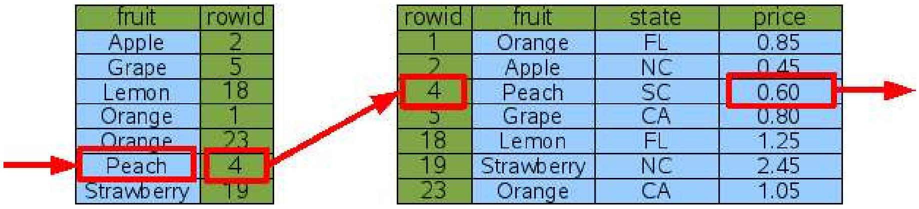
fruit	rowid
Apple	2
Grape	5
Lemon	18
Orange	1
Orange	23
Peach	4
Strawberry	19

Figure 4: An Index On The Fruit Column

This new index can be used to implement a faster algorithm for the original "Price of Peaches" query.

```
SELECT price FROM fruitsforsale WHERE fruit='Peach';
```

The query starts by doing a binary search on the Idx1 index for entries that have fruit='Peach'. SQLite can do this binary search on the Idx1 index but not on the original FruitsForSale table because the rows in Idx1 are sorted by the "fruit" column. Having found a row in the Idx1 index that has fruit='Peach', the database engine can extract the rowid for that row. Then the database engines does a second binary search on the original FruitsForSale table to find the original row that contains fruit='Peach'. From the row in the FruitsForSale table, SQLite can then extract the value of the price column. This procedure is illustrated by figure 5.



fruit	rowid
Apple	2
Grape	5
Lemon	18
Orange	1
Orange	23
Peach	4
Strawberry	19

rowid	fruit	state	price
1	Orange	FL	0.85
2	Apple	NC	0.45
4	Peach	SC	0.60
5	Grape	CA	0.80
18	Lemon	FL	1.25
19	Strawberry	NC	2.45
23	Orange	CA	1.05

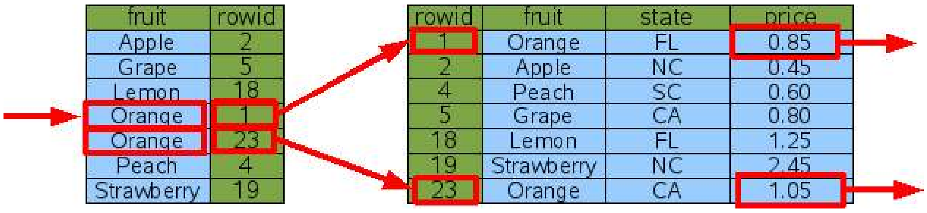
Figure 5: Indexed Lookup For The Price Of Peaches

SQLite has to do two binary searches to find the price of peaches using the method show above. But for a table with a large number of rows, this is still much faster than doing a full table scan.

1.4. Multiple Result Rows

In the previous query the fruit='Peach' constraint narrowed the result down to a single row. But the same technique works even if multiple rows are obtained. Suppose we looked up the price of Oranges instead of Peaches:

```
SELECT price FROM fruitsforsale WHERE fruit='Orange'
```



fruit	rowid
Apple	2
Grape	5
Lemon	18
Orange	1
Orange	23
Peach	4
Strawberry	19

rowid	fruit	state	price
1	Orange	FL	0.85
2	Apple	NC	0.45
4	Peach	SC	0.60
5	Grape	CA	0.80
18	Lemon	FL	1.25
19	Strawberry	NC	2.45
23	Orange	CA	1.05

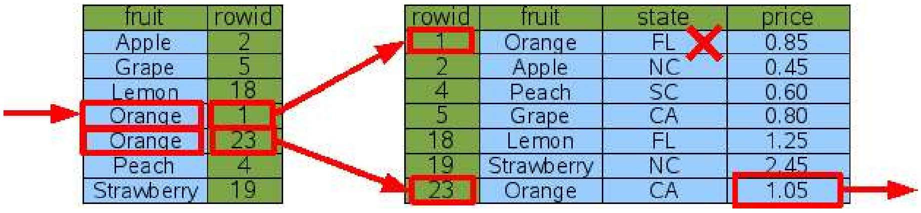
Figure 6: Indexed Lookup For The Price Of Oranges

In this case, SQLite still does a single binary search to find the first entry of the index where fruit='Orange'. Then it extracts the rowid from the index and uses that rowid to lookup the original table entry via binary search and output the price from the original table. But instead of quitting, the database engine then advances to the next row of index to repeat the process for next fruit='Orange' entry. Advancing to the next row of an index (or table) is much less costly than doing a binary search since the next row is often located on the same database page as the current row. In fact, the cost of advancing to the next row is so cheap in comparison to a binary search that we usually ignore it. So our estimate for the total cost of this query is 3 binary searches. If the number of rows of output is K and the number of rows in the table is N, then in general the cost of doing the query is proportional to (K+1)*logN.

1.5. Multiple AND-Connected WHERE-Clause Terms

Next, suppose that you want to look up the price of not just any orange, but specifically California-grown oranges. The appropriate query would be as follows:

```
SELECT price FROM fruitsforsale WHERE fruit='Orange' AND state='CA'
```



fruit	rowid
Apple	2
Grape	5
Lemon	18
Orange	1
Orange	23
Peach	4
Strawberry	19

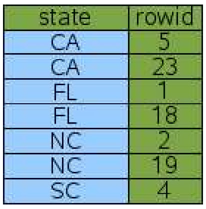
rowid	fruit	state	price
1	Orange	FL	0.85
2	Apple	NC	0.45
4	Peach	SC	0.60
5	Grape	CA	0.80
18	Lemon	FL	1.25
19	Strawberry	NC	2.45
23	Orange	CA	1.05

Figure 7: Indexed Lookup Of California Oranges

One approach to this query is to use the fruit='Orange' term of the WHERE clause to find all rows dealing with oranges, then filter those rows by rejecting any that are from states other than California. This process is shown by figure 7 above. This is a perfectly reasonable approach in most cases. Yes, the database engine did have to do an extra binary search for the Florida orange row that was later rejected, so it was not as efficient as we might hope, though for many applications it is efficient enough.

Suppose that in addition to the index on "fruit" there was also an index on "state".

```
CREATE INDEX Idx2 ON fruitsforsale(state);
```

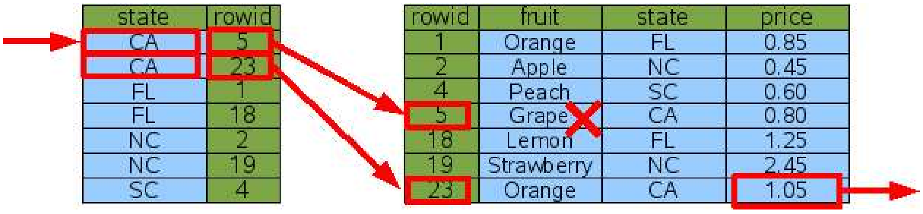


state	rowid
CA	5
CA	23
FL	1
FL	18
NC	2
NC	19
SC	4

Figure 8: Index On The State Column

The "state" index works just like the "fruit" index in that it is a new table with an extra column in front of the rowid and sorted by that extra column as the primary key. The only difference is that in Idx2, the first column is "state" instead of "fruit" as it is with Idx1. In our example data set, there is more redundancy in the "state" column and so they are more duplicate entries. The ties are still resolved using the rowid.

Using the new Idx2 index on "state", SQLite has another option for lookup up the price of California oranges: it can look up every row that contains fruit from California and filter out those rows that are not oranges.



state	rowid
CA	5
CA	23
FL	1
FL	18
NC	2
NC	19
SC	4

rowid	fruit	state	price
1	Orange	FL	0.85
2	Apple	NC	0.45
4	Peach	SC	0.60
5	Grape	CA	0.80
18	Lemon	FL	1.25
19	Strawberry	NC	2.45
23	Orange	CA	1.05

Figure 9: Indexed Lookup Of California Oranges

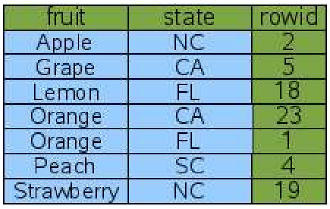
Using Idx2 instead of Idx1 causes SQLite to examine a different set of rows, but it gets the same answer in the end (which is very important - remember that indices should never change the answer, only help SQLite to get to the answer more quickly) and it does the same amount of work. So the Idx2 index did not help performance in this case.

The last two queries take the same amount of time, in our example. So which index, Idx1 or Idx2, will SQLite choose? If the ANALYZE command has been run on the database, so that SQLite has had an opportunity to gather statistics about the available indices, then SQLite will know that the Idx1 index usually narrows the search down to a single item (our example of fruit='Orange' is the exception to this rule) whereas the Idx2 index will normally only narrow the search down to two rows. So, if all else is equal, SQLite will choose Idx1 with the hope of narrowing the search to as small a number of rows as possible. This choice is only possible because of the statistics provided by ANALYZE. If ANALYZE has not been run then the choice of which index to use is arbitrary.

1.6. Multi-Column Indices

To get the maximum performance out of a query with multiple AND-connected terms in the WHERE clause, you really want a multi-column index with columns for each of the AND terms. In this case we create a new index on the "fruit" and "state" columns of FruitsForSale:

```
CREATE INDEX Idx3 ON FruitsForSale(fruit, state);
```



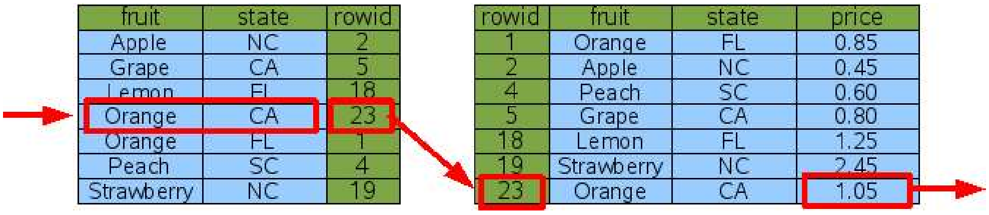
fruit	state	rowid
Apple	NC	2
Grape	CA	5
Lemon	FL	18
Orange	CA	23
Orange	FL	1
Peach	SC	4
Strawberry	NC	19

Figure 1: A Two-Column Index

A multi-column index follows the same pattern as a single-column index; the indexed columns are added in front of the rowid. The only difference is that now multiple columns are added. The left-most column is the primary key used for ordering the rows in the index. The second column is used to break ties in the left-most column. If there were a third column, it would be used to break ties for the first two columns. And so forth for all columns in the index. Because rowid is guaranteed to be unique, every row of the index will be unique even if all of the content columns for two rows are the same. That case does not happen in our sample data, but there is one case (fruit='Orange') where there is a tie on the first column which must be broken by the second column.

Given the new multi-column Idx3 index, it is now possible for SQLite to find the price of California oranges using only 2 binary searches:

```
SELECT price FROM fruitsforsale WHERE fruit='Orange' AND state='CA'
```



fruit	state	rowid
Apple	NC	2
Grape	CA	5
Lemon	FL	18
Orange	CA	23
Orange	FL	1
Peach	SC	4
Strawberry	NC	19

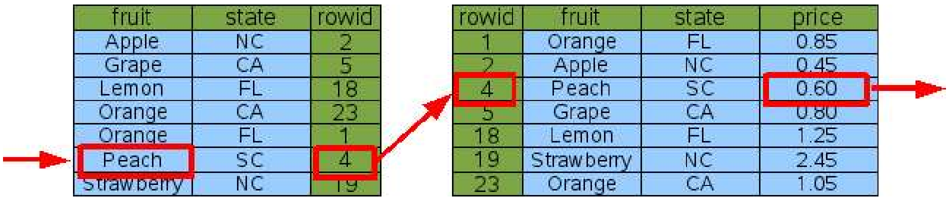
rowid	fruit	state	price
1	Orange	FL	0.85
2	Apple	NC	0.45
4	Peach	SC	0.60
5	Grape	CA	0.80
18	Lemon	FL	1.25
19	Strawberry	NC	2.45
23	Orange	CA	1.05

Figure 11: Lookup Using A Two-Column Index

With the Idx3 index on both columns that are constrained by the WHERE clause, SQLite can do a single binary search against Idx3 to find the one rowid for California oranges, then do a single binary search to find the price for that item in the original table. There are no dead-ends and no wasted binary searches. This is a more efficient query.

Note that Idx3 contains all the same information as the original Idx1. And so if we have Idx3, we do not really need Idx1 any more. The "price of peaches" query can be satisfied using Idx3 by simply ignoring the "state" column of Idx3:

```
SELECT price FROM fruitsforsale WHERE fruit='Peach'
```



fruit	state	rowid
Apple	NC	2
Grape	CA	5
Lemon	FL	18
Orange	CA	23
Orange	FL	1
Peach	SC	4
Strawberry	NC	19

rowid	fruit	state	price
1	Orange	FL	0.85
2	Apple	NC	0.45
4	Peach	SC	0.60
5	Grape	CA	0.80
18	Lemon	FL	1.25
19	Strawberry	NC	2.45
23	Orange	CA	1.05

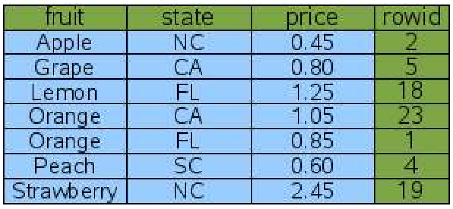
Figure 12: Single-Column Lookup On A Multi-Column Index

Hence, a good rule of thumb is that your database schema should never contain two indices where one index is a prefix of the other. Drop the index with fewer columns. SQLite will still be able to do efficient lookups with the longer index.

1.7. Covering Indexes

The "price of California oranges" query was made more efficient through the use of a two-column index. But SQLite can do even better with a three-column index that also includes the "price" column:

```
CREATE INDEX Idx4 ON FruitsForSale(fruit, state, price);
```

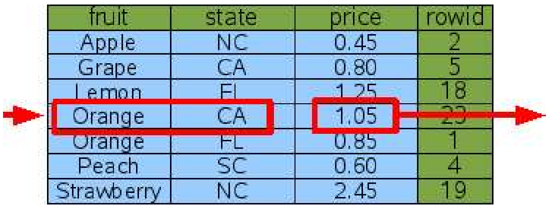


fruit	state	price	rowid
Apple	NC	0.45	2
Grape	CA	0.80	5
Lemon	FL	1.25	18
Orange	CA	1.05	23
Orange	FL	0.85	1
Peach	SC	0.60	4
Strawberry	NC	2.45	19

Figure 13: A Covering Index

This new index contains all the columns of the original FruitsForSale table that are used by the query - both the search terms and the output. We call this a "covering index". Because all of the information needed is in the covering index, SQLite never needs to consult the original table in order to find the price.

```
SELECT price FROM fruitsforsale WHERE fruit='Orange' AND state='CA';
```



fruit	state	price	rowid
Apple	NC	0.45	2
Grape	CA	0.80	5
Lemon	FL	1.25	18
Orange	CA	1.05	23
Orange	FL	0.85	1
Peach	SC	0.60	4
Strawberry	NC	2.45	19

Figure 14: Query Using A Covering Index

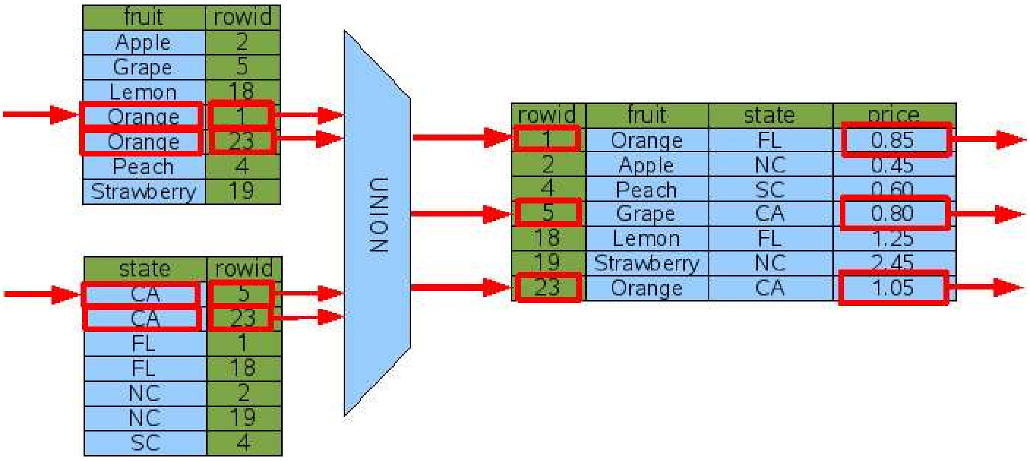
Hence, by adding extra "output" columns onto the end of an index, one can avoid having to reference the original table and thereby cut the number of binary searches for a query in half. This is a constant-factor improvement in performance (roughly a doubling of the speed). But on the other hand, it is also just a refinement; A two-fold performance increase is not nearly as dramatic as the one-million-fold increase seen when the table was first indexed. And for most queries, the difference between 1 microsecond and 2 microseconds is unlikely to be noticed.

1.8. OR-Connected Terms In The WHERE Clause

Multi-column indices only work if the constraint terms in the WHERE clause of the query are connected by AND. So Idx3 and Idx4 are helpful when the search is for items that are both Oranges and grown in California, but neither index would be that useful if we wanted all items that were either oranges *or* are grown in California.

```
SELECT price FROM FruitsForSale WHERE fruit='Orange' OR state='CA';
```

When confronted with OR-connected terms in a WHERE clause, SQLite examines each OR term separately and tries to use an index to find the rowids associated with each term. It then takes the union of the resulting rowid sets to find the end result. The following figure illustrates this process:



fruit	rowid
Apple	2
Grape	5
Lemon	18
Orange	1
Orange	23
Peach	4
Strawberry	19

state	rowid
CA	5
CA	23
FL	1
FL	18
NC	2
NC	19
SC	4

rowid	fruit	state	price
1	Orange	FL	0.85
2	Apple	NC	0.45
4	Peach	SC	0.60
5	Grape	CA	0.80
18	Lemon	FL	1.25
19	Strawberry	NC	2.45
23	Orange	CA	1.05

Figure 15: Query With OR Constraints

The diagram above implies that SQLite computes all of the rowids first and then combines them with a union operation before starting to do rowid lookups on the original table. In reality, the rowid lookups are interspersed with rowid computations. SQLite uses one index at a time to find rowids while remembering which rowids it has seen before so as to avoid duplicates. That is just an implementation detail, though. The diagram, while not 100% accurate, provides a good overview of what is happening.

In order for the OR-by-UNION technique shown above to be useful, there must be an index available that helps resolve every OR-connected term in the WHERE clause. If even a single OR-connected term is not indexed, then a full table scan would have to be done in order to find the rowids generated by the one term, and if SQLite has to do a full table scan, it might as well do it on the original table and get all of the results in a single pass without having to mess with union operations and follow-on binary searches.

One can see how the OR-by-UNION technique could also be leveraged to use multiple indices on queries where the WHERE clause has terms connected by AND, by using an intersect operator in place of union. Many SQL database engines will do just that. But the performance gain over using just a single index is slight and so SQLite does not implement that technique at this time. However, a future version SQLite might be enhanced to support AND-by-INTERSECT.

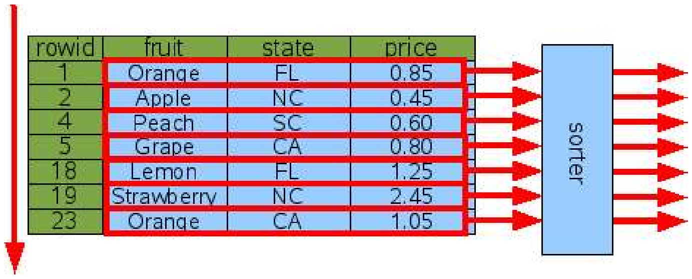
2. Sorting

SQLite (like all other SQL database engines) can also use indices to satisfy the ORDER BY clauses in a query, in addition to expediting lookup. In other words, indices can be used to speed up sorting as well as searching.

When no appropriate indices are available, a query with an ORDER BY clause must be sorted as a separate step. Consider this query:

```
SELECT * FROM fruitsforsale ORDER BY fruit;
```

SQLite processes this by gathering all the output of query and then running that output through a sorter.



rowid	fruit	state	price
1	Orange	FL	0.85
2	Apple	NC	0.45
4	Peach	SC	0.60
5	Grape	CA	0.80
18	Lemon	FL	1.25
19	Strawberry	NC	2.45
23	Orange	CA	1.05

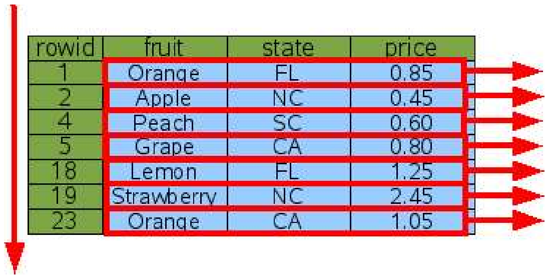
Figure 16: Sorting Without An Index

If the number of output rows is K, then the time needed to sort is proportional to KlogK. If K is small, the sorting time is usually not a factor, but in a query such as the above where K==N, the time needed to sort can be much greater than the time needed to do a full table scan. Furthermore, the entire output is accumulated in temporary storage (which might be either in main memory or on disk, depending on various compile-time and run-time settings) which can mean that a lot of temporary storage is required to complete the query.

2.1. Sorting By Rowid

Because sorting can be expensive, SQLite works hard to convert ORDER BY clauses into no-ops. If SQLite determines that output will naturally appear in the order specified, then no sorting is done. So, for example, if you request the output in rowid order, no sorting will be done:

```
SELECT * FROM fruitsforsale ORDER BY rowid;
```



rowid	fruit	state	price
1	Orange	FL	0.85
2	Apple	NC	0.45
4	Peach	SC	0.60
5	Grape	CA	0.80
18	Lemon	FL	1.25
19	Strawberry	NC	2.45
23	Orange	CA	1.05

Figure 17: Sorting By Rowid

You can also request a reverse-order sort like this:

```
SELECT * FROM fruitsforsale ORDER BY rowid DESC;
```

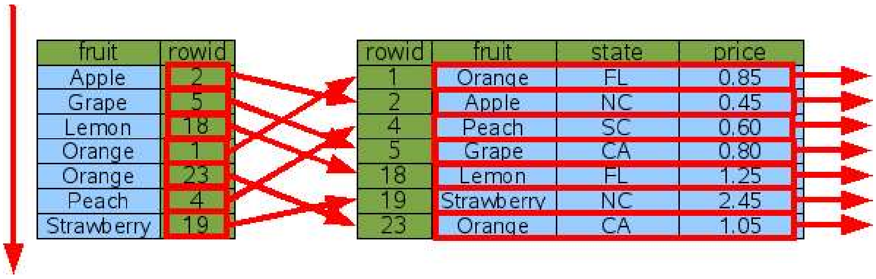
SQLite will still omit the sorting step. But in order for output to appear in the correct order, SQLite will do the table scan starting at the end and working toward the beginning, rather than starting at the beginning and working toward the end as shown in figure 17.

2.2. Sorting By Index

Of course, ordering the output of a query by rowid is seldom useful. Usually one wants to order the output by some other column.

If an index is available on the ORDER BY column, that index can be used for sorting. Consider the request for all items sorted by "fruit":

```
SELECT * FROM fruitsforsale ORDER BY fruit;
```



fruit	rowid
Apple	2
Grape	5
Lemon	18
Orange	1
Orange	23
Peach	4
Strawberry	19

rowid	fruit	state	price
1	Orange	FL	0.85
2	Apple	NC	0.45
4	Peach	SC	0.60
5	Grape	CA	0.80
18	Lemon	FL	1.25
19	Strawberry	NC	2.45
23	Orange	CA	1.05

Figure 18: Sorting With An Index

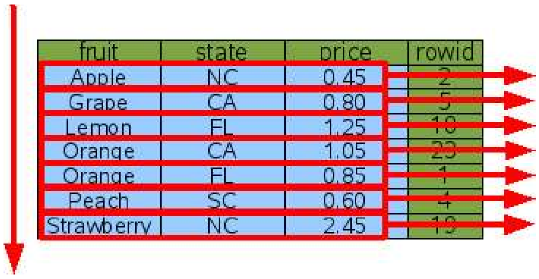
The Idx1 index is scanned from top to bottom (or from bottom to top if "ORDER BY fruit DESC" is used) in order to find the rowids for each item in order by fruit. Then for each rowid, a binary search is done to lookup and output that row. In this way, the output appears in the requested order without the need to gather the entire output and sort it using a separate step.

But does this really save time? The number of steps in the original indexless sort is proportional to NlogN since that is how much time it takes to sort N rows. But when we use Idx1 as shown here, we have to do N rowid lookups which take logN time each, so the total time of NlogN is the same!

SQLite uses a cost-based query planner. When there are two or more ways of solving the same query, SQLite tries to estimate the total amount of time needed to run the query using each plan, and then uses the plan with the lowest estimated cost. A cost is computed mostly from the estimated time, and so this case could go either way depending on the table size and what WHERE clause constraints were available, and so forth. But generally speaking, the indexed sort would probably be chosen, if for no other reason, because it does not need to accumulate the entire result set in temporary storage before sorting and thus uses much less temporary storage.

2.3. Sorting By Covering Index

If a covering index can be used for a query, then the multiple rowid lookups can be avoided and the cost of the query drops dramatically.



fruit	state	price	rowid
Apple	NC	0.45	2
Grape	CA	0.80	5
Lemon	FL	1.25	18
Orange	CA	1.05	23
Orange	FL	0.85	1
Peach	SC	0.60	4
Strawberry	NC	2.45	19

Figure 19: Sorting With A Covering Index

With a covering index, SQLite can simply walk the index from one end to the other and deliver the output in time proportional to N and without having allocate a large buffer to hold the result set.

3. Searching And Sorting At The Same Time

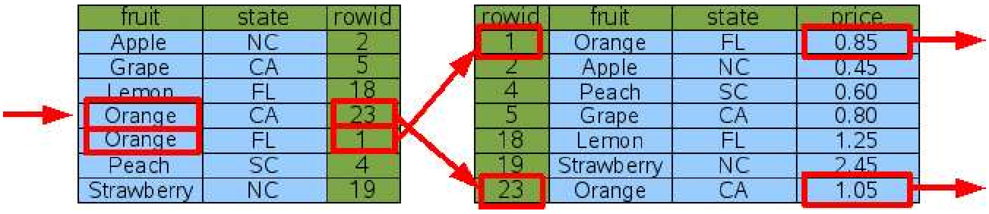
The previous discussion has treated searching and sorting as separate topics. But in practice, it is often the case that one wants to search and sort at the same time. Fortunately, it is possible to do this using a single index.

3.1. Searching And Sorting With A Multi-Column Index

Suppose we want to find the prices of all kinds of oranges sorted in order of the state where they are grown. The query is this:

```
SELECT price FROM fruitforsale WHERE fruit='Orange' ORDER BY state
```

The query contains both a search restriction in the WHERE clause and a sort order in the ORDER BY clause. Both the search and the sort can be accomplished at the same time using the two-column index Idx3.



fruit	state	rowid
Apple	NC	2
Grape	CA	5
Lemon	FL	18
Orange	CA	23
Orange	FL	1
Peach	SC	4
Strawberry	NC	19

rowid	fruit	state	price
1	Orange	FL	0.85
2	Apple	NC	0.45
4	Peach	SC	0.60
5	Grape	CA	0.80
18	Lemon	FL	1.25
19	Strawberry	NC	2.45
23	Orange	CA	1.05

Figure 20: Search And Sort By Multi-Column Index

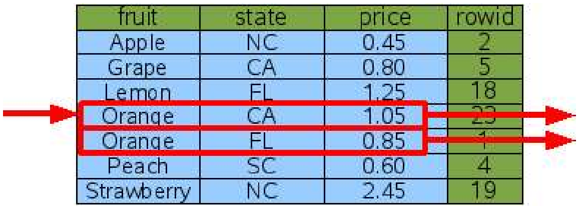
The query does a binary search on the index to find the subset of rows that have fruit='Orange'. (Because the fruit column is the left-most column of the index and the rows of the index are in sorted order, all such rows will be adjacent.) Then it scans the matching index rows from top to bottom to get the rowids for the original table, and for each rowid does a binary search on the original table to find the price.

You will notice that there is no "sort" box anywhere in the above diagram. The ORDER BY clause of the query has become a no-op. No sorting has to be done here because the output order is by the state column and the state column also happens to be the first column after the fruit column in the index. So, if we scan entries of the index that have the same value for the fruit column from top to bottom, those index entries are guaranteed to be ordered by the state column.

3.2. Searching And Sorting With A Covering Index

A covering index can also be used to search and sort at the same time. Consider the following:

```
SELECT * FROM fruitforsale WHERE fruit='Orange' ORDER BY state
```



fruit	state	price	rowid
Apple	NC	0.45	2
Grape	CA	0.80	5
Lemon	FL	1.25	18
Orange	CA	1.05	23
Orange	FL	0.85	1
Peach	SC	0.60	4
Strawberry	NC	2.45	19

Figure 21: Search And Sort By Covering Index

As before, SQLite does single binary search for the range of rows in the covering index that satisfy the WHERE clause, the scans that range from top to bottom to get the desired results. The rows that satisfy the WHERE clause are guaranteed to be adjacent since the WHERE clause is an equality constraint on the left-most column of the index. And by scanning the matching index rows from top to bottom, the output is guaranteed to be ordered by state since the state column is the very next column to the right of the fruit column. And so the resulting query is very efficient.

SQLite can pull a similar trick for a descending ORDER BY:

```
SELECT * FROM fruitforsale WHERE fruit='Orange' ORDER BY state DESC
```

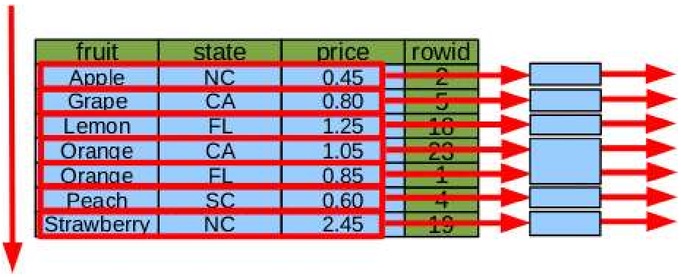
The same basic algorithm is followed, except this time the matching rows of the index are scanned from bottom to top instead of from top to bottom, so that the states will appear in descending order.

3.3. Partial Sorting Using An Index (a.k.a. Block Sorting)

Sometimes only part of an ORDER BY clause can be satisfied using indexes. Consider, for example, the following query:

```
SELECT * FROM fruitforsale ORDER BY fruit, price
```

If the covering index is used for the scan, the "fruit" column will appear naturally in the correct order, but when there are two or more rows with the same fruit, the price might be out of order. When this occurs, SQLite does many small sorts, one sort for each distinct value of fruit, rather than one large sort. Figure 22 below illustrates the concept.



fruit	state	price	rowid
Apple	NC	0.45	2
Grape	CA	0.80	5
Lemon	FL	1.25	18
Orange	CA	1.05	23
Orange	FL	0.85	1
Peach	SC	0.60	4
Strawberry	NC	2.45	19

Figure 22: Partial Sort By Index

In the example, instead of a single sort of 7 elements, there are 5 sorts of one-element each and 1 sort of 2 elements for the case of fruit=='Orange'.

The advantages of doing many smaller sorts instead of a single large sort are:

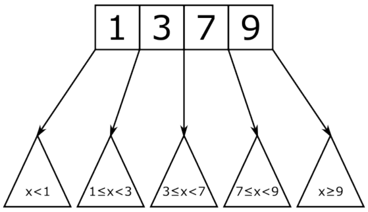
1. Multiple small sorts collectively use fewer CPU cycles than a single large sort.
2. Each small sort is run independently, meaning that much less information needs to be kept in temporary storage at any one time.
3. Those columns of the ORDER BY that are already in the correct order due to indexes can be omitted from the sort key, further reducing storage requirements and CPU time.
4. Output rows can be returned to the application as each small sort completes, and well before the table scan is complete.
5. If a LIMIT clause is present, it might be possible to avoid scanning the entire table.

Because of these advantages, SQLite always tries to do a partial sort using an index even if a complete sort by index is not possible.

This page last modified on 2022-10-26 13:30:36 UTC

8.3 B^+ Trees

- Indices can be too large to load into memory at once, making IO operations and binary searches for non-extreme values expensive
- This can be resolved by storing the index as a multilevel index, where the actual contents of the index are stored in the terminal nodes of a tree
  - ↳ The internal nodes allow for the correct data block/terminal node to be searched for and identified
- A binary tree suffers from several problems, which are ultimately resolved by using a B^+ tree:
  - ↳ There are many internal nodes, so that $\log_2(N)$ nodes must be read in order to find the desired key
  - ↳ The next highest key (useful for key ranges) is not stored in memory near the current key, requiring more IO operations
  - ↳ Binary trees can become unbalanced or require expensive restructuring in order to remain balanced
- An M-ary search tree is an extension of a binary tree but with a maximum branching factor of M rather than 2
  - ↳ In a complete tree, the depth is $\log_M(N)$, each internal node has $M-1$ keys, and the expected runtime of a lookup is $M\log_M(N)$
- A B^+ tree is a specified M-ary search tree where each internal node contains $M-1$ keys
  - ↳ Each internal node other than the root has between $\lceil M/2 \rceil$ and M children
  - ↳ The internal node contains one fewer search key than children, so that the search keys mark the "boundaries" between the children
  - ↳ Each leaf node contains a block of data, between $\lceil L/2 \rceil$ and L key values



An internal node of a B^+ tree with $M = 5$

  - ↳ Binary searches are used when traversing the tree, both with internal nodes and leaf nodes, allowing for asymptotic runtime to be logarithmic
  - ↳ For a B^+ tree to be used to store an index, each leaf node should point to the subsequent leaf node so that it can be easily traversed
- When inserting into a B^+ tree, perform the following:
  - Identify the proper leaf node by traversing the tree then insert
  - If the leaf has $L+1$ items, split the leaf into two nodes, of sizes $\lceil (L+1)/2 \rceil$ and $\lfloor (L+1)/2 \rfloor$. If the parent has $M+1$ children, perform following step
  - If an internal non-root node has $M+1$ children, split the node into two nodes, one with $\lceil (M+1)/2 \rceil$ children and the other with $\lfloor (M+1)/2 \rfloor$. If the parent has $M+1$ children, repeat this step on that node if it is not the root, or perform the following step
  - If the root node has $M+1$ children, make the tree deeper by splitting the root node in half (as above) then creating a new root node whose children are these two halves
- When deleting from a B^+ tree, perform the following:

  - Identify the proper leaf node by traversing the tree then delete
  - If the leaf note has fewer than $\lceil L/2 \rceil$ items, then if either neighbor has extra items, adopt one and update the parent
  - If no neighbor has extra items, then divide up the node's items with its neighbor(s), update the parent, then delete the node
  - If the previous step causes the parent to have fewer than $\lceil M/2 \rceil$ children, then repeat the process, either adopting neighboring children or deleting
  - If the previous step causes the root to have only one child, then replace the root with its child

- Insertion can cause expensive splitting with propagation, while deletion can cause either cheap adoption or expensive deletion with propagation
  - ↳ These expensive operations are rare with M and L large, such as $M = L = 128$

8.4 Alternate Index Structures

- If lookup ranges are rare, then hash tables can be used
  - ↳ For a good hash function, lookup is nearly constant time, allowing for quicker performance than B$^+$ trees
  - ↳ However, hash tables are unsorted, so they do not work for range lookups
  - ↳ Performance also decreases as the hash table fills up
- If keys in the index are often duplicated, bitmaps can provide better performance, especially for compound operations like AND/OR/NOT
  - ↳ Each key has its own associated bitmap, such as in the example:

rowid	student_id	first_name	major	honors
0	1923	Alice	CSE	1
1	3451	Bob	MTH	0
2	2323	Carol	Other	0
3	4561	David	Other	1
4	9365	Emily	CSE	0
5	1086	Frank	MTH	1

`major_CSE`	100010
`major_MTH`	010001
`honors_1`	100101

  - ↳ A query selecting CSE majors or MTH majors, neither with honors, would use bitmap operations to yield the bitmap 010010, corresponding to the rowid 1 and 4
- Space efficiency is high assuming that there are a few highly duplicated keys, lookups with multiple criteria are efficient, but range lookups are not efficient
- Insertion and deletion can be highly expensive, often requiring the bitmap to be regenerated
  - ↳ To lessen this effect, and existence bitmap can track whether a row has been deleted or not, and then this can be implicitly added to the criteria of queries