

Contents

1	File Types	2
1.1	CSV Files	2
1.2	JSON Files	2
1.3	XML File	3
2	Database Concepts	5
2.1	Introduction	5
2.2	CAP Theorem for Distributed Databases	5
3	Entity-Relationship Model	6
3.1	Introduction	6
3.2	Subclasses	7
3.3	Keys	8
3.4	Design Decisions	9
3.5	Converting E/R Diagrams to Relations	10
3.6	Determining Decompositions	10
4	Transaction Management	11
4.1	Introduction	11
4.2	Serializable Schedules	11
4.3	Conflict-Serializable Schedules	11
4.4	Locks	12
4.5	Types of Locks	13
4.6	Transactions in SQLite	13
4.7	Rollbacks	14
4.8	Deadlock	14
4.9	Validation Scheduling	15
4.10	Timestamp-Based Scheduling	16
4.11	Comparison of Concurrency Control Methods	17
5	Relational Database Model	18
5.1	Structure of Relational Databases	18
6	SQL	19
6.1	Introduction to SQL	19
6.2	SQL Style	19
6.3	SQLite Datatypes	19
6.4	Adding Data	20
6.5	Updating Data	21
6.6	Removing Data	21
6.7	Querying	21
6.8	SQL Operators	22
6.9	SQL Functions	22
6.10	SQL Aggregators	22
6.11	Collations	23
6.12	Case Expression	23
6.13	Column Constraints	23
6.14	Table Constraints	24
6.15	Qualified Table Names	24
6.16	SQL Union	25
6.17	SQL Joins	25
6.18	SQL Subqueries	26
6.19	Group By	27
6.20	Dates and Times	27
6.21	Stored Procedures	28
6.22	Triggers	29

6.23 DDL Operations	30
6.24 Views	31
6.25 Indices	31
6.26 Miscellaneous SQL	31
Other Notes	31
Query Planning/Indices	31

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
- The types in JSON are:
 - **"string"**, a sequence of zero or more characters enclosed in quotation marks
 - ↳ 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"**
 - **"boolean"**, which is either **true**, **false**
 - **null**
- A sample JSON schema is:

```

1 {
2   "title": "my_title",
3   "description": "my_description",
4   "type": "object"
5   "properties": {
6     "field1": {
7       "description": "description1",
8       "type": "type1"
9     },
10    "field2": {
11      "description": "description2",
12      "type": "type2"
13    }
14  },
15  "required": ["field1", "field2"]
16 }

```

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 columns `<tag_name attr_name="value">`
 - ↳ Only one value may be specified, so if multiple are needed, oftentimes space-delimited strings are used
- An XML schema uses the following format:

```

1 <xsd:schema xmlns:xsd="schema_link">
2 <xsd:annotation>
3   <xsd:documentation xml:lang="en">
4     Description Here
5   </xsd:documentation>
6 </xsd:annotation>
7 <xsd:element name="name1" type="myType1"/>
8 <xsd:complexType name="myType1">
9   <xsd:sequence>
10    <xsd:element name="name2" type="xsd:type1"/>
11    <xsd:element name="name3" type="xsd:type2"/>
12  </xsd:sequence>
13 </xsd:complexType>
14 </xsd:schema>

```

- The built-in simple types are "xsd:string", "xsd:decimal", "xsd:integer", "xsd:boolean", "xsd:date", "xsd:time", "xsd:dateTime", and "xsd:duration"
- Other attributes include `default`, `use` (set equal to "required" to make required)

- More detailed restrictions can be placed by defining a simple type, such as:

```
1 <xsd:simpleType name="myType">
2   <xsd:restriction base="xsd:integer">
3     <xsd:minInclusive value="0"/>
4     <xsd:maxInclusive value="120"/>
5   </xsd:restriction>
6 </xsd:simpleType>
```

2. Database Concepts

2.1 Introduction

- 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 means the data conforms to the schema (and is well-formed)

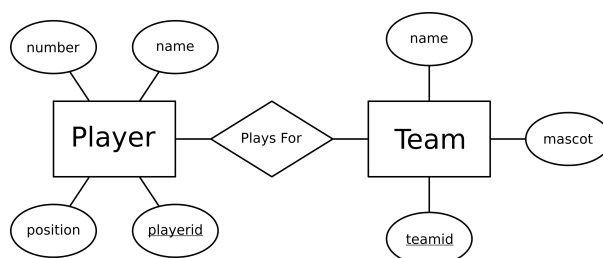
2.2 CAP Theorem for Distributed Databases

- A distributed database stores information on many nodes or virtual machines
- 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 than 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
- This theorem yields the following cases:
 - CA:
 - CP:
 - AP:

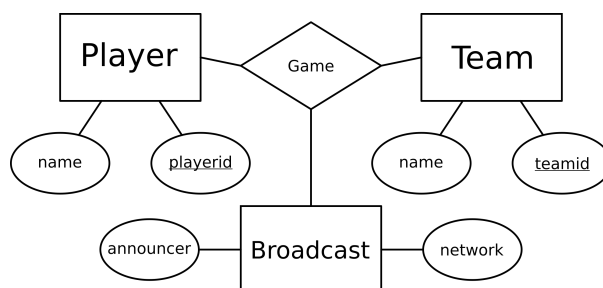
3. Entity-Relationship Model

3.1 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

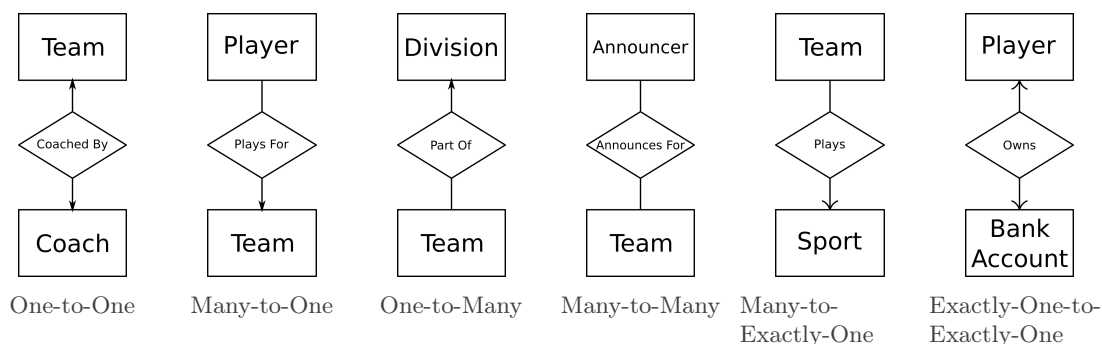
Value of Teams Entity Set

playerid	teamid
0	4
1	9
2	15

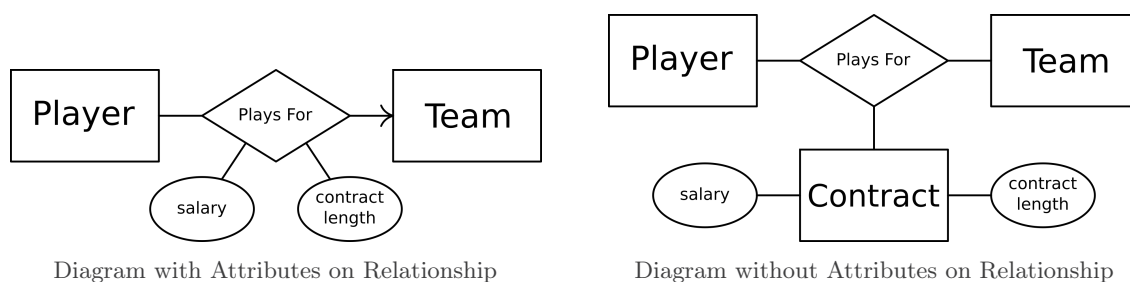
Value of Plays For Relationship Set

- 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

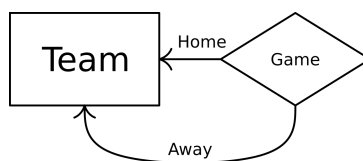
- If one entity is related to exactly one of another entity, this is represent with a rounded arrow



- Attributes can be present on a relationship:



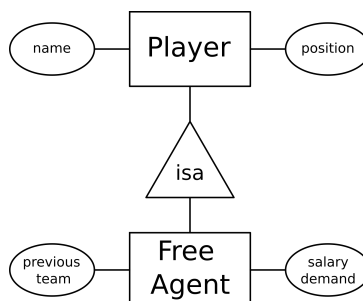
- Edges are sometimes labeled with roles to disambiguate the case where an entity set appears multiple times in a relationship



3.2 Subclasses

- A subclass is a special case of an entity set with more properties (either attributes or relationships)

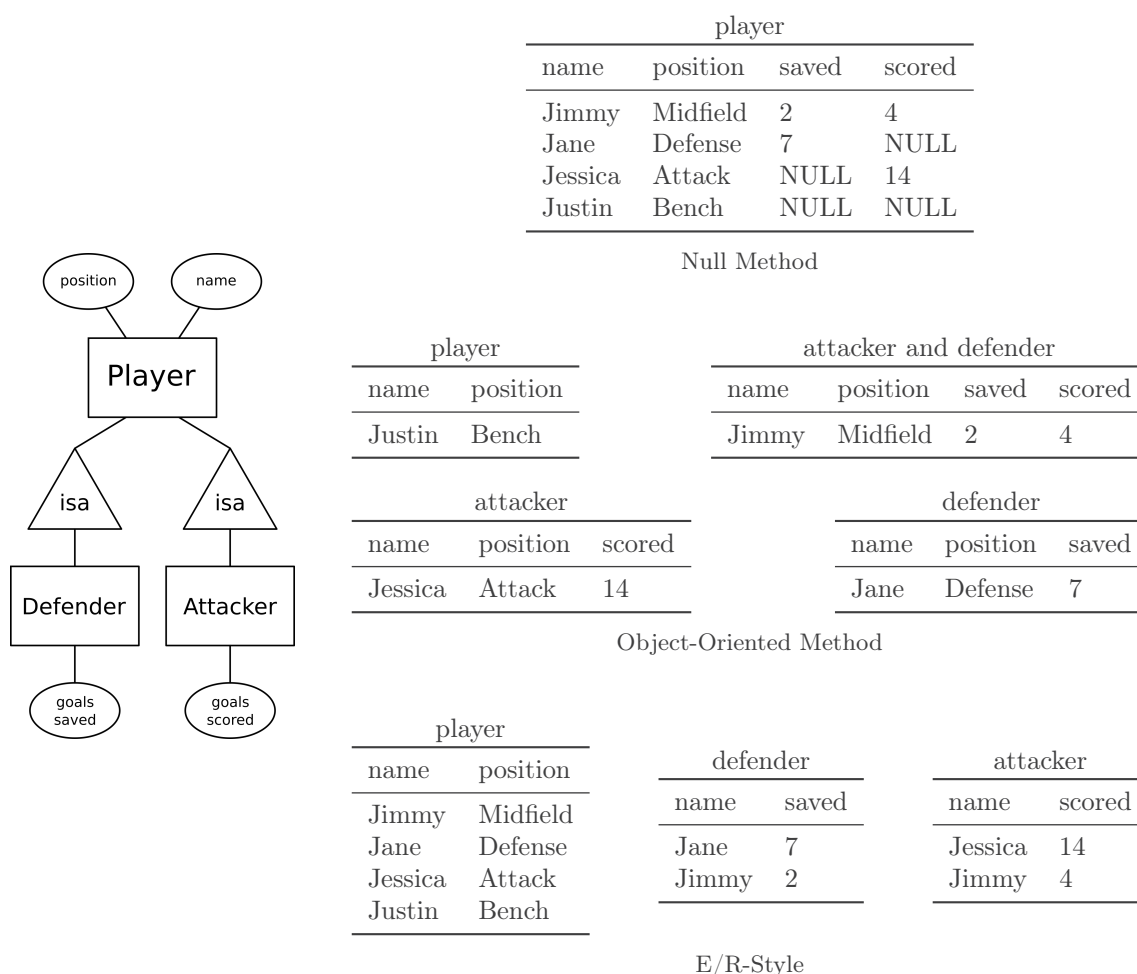
↳ Note that no multiple inheritance is allowed



- 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.

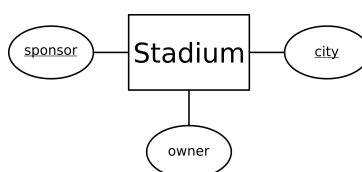
(iii) E/R-Style: Use one relation for each subclass with attributes of that subclass and key attributes. List an object in the subclass relation and all superclass relations.

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

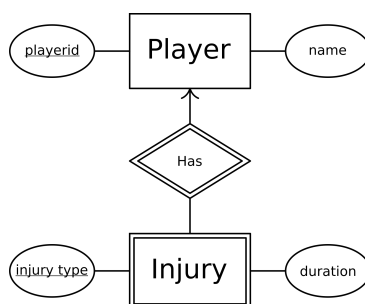


3.3 Keys

- 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

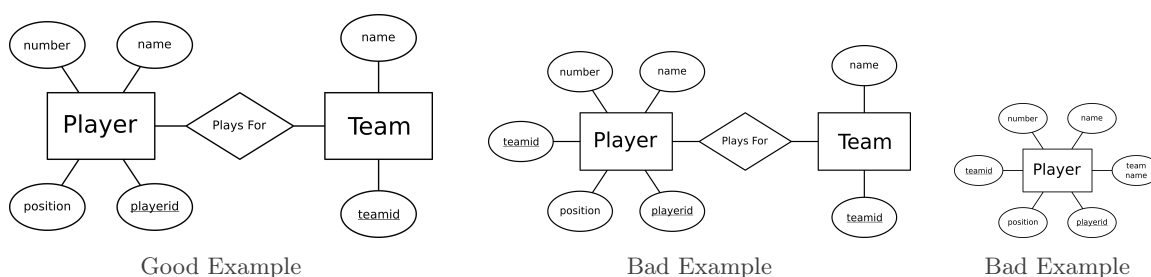


- A weak entity E is one such that the key of other entity sets needs to be included in order to uniquely identify entries, by following at least one many-to-exactly-one relationship from E
 - ↳ This is denoted by adding a double outline around the weak identity and supporting relationships



3.4 Design Decisions

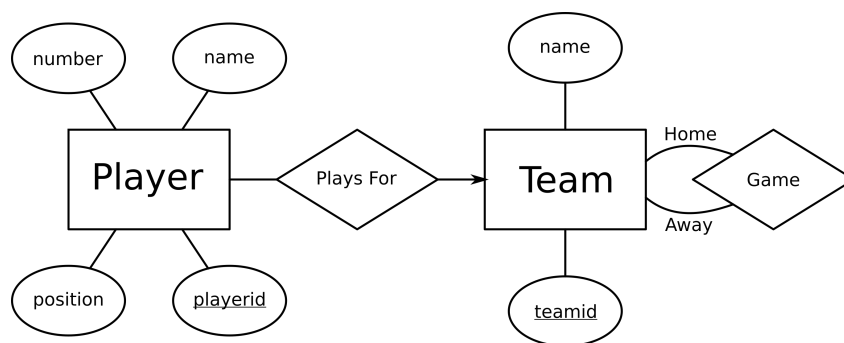
- Consider all the data in a database stored in a single table, with null values and redundancy. Decompositions of this table are made when it is divided into smaller relations/entity sets. All decompositions should be lossless, so that the original table could be recovered via natural joins on the subrelations
- A database should avoid redundancy, both to limit space requirements and to reduce the risk of inconsistency
 - 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 group project when a single member of the group drops the class)



- An entity set should be used instead of an attribute only when either:
 - The quantity is more than just the name of something; it has at least one non-key attribute
 - 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, automobile VINs, etc.
 - Weak entities are only needed when there is no global authority capable of establishing a unique identifier

3.5 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
- 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)

3.6 Determining Decompositions

- As described, having too large of relations can cause anomalies; to solve this, we develop a theory for determining whether a relation is in good form or needs to be decomposed into subrelations
- To determine when decompositions are possible, we introduce functional dependency, which captures information about the constraints in a database
 - ↳ A set of attributes Y is functionally dependent on a set of attributes X in relation R , denoted $X \rightarrow Y$, if any rows agreeing on the values in X also agree on the values in Y
 - ↳ $X \rightarrow Y$ 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
 - ↳ $X \rightarrow A_1 A_2 \cdots A_n$ is equivalent to $X \rightarrow A_i$ for all $1 \leq i \leq n$ (this is the splitting/combining rule)
 - ↳ A functional dependency $X \rightarrow Y$ is trivial if $Y \subset X$
- Using functional dependency, we can revisit keys
 - ↳ K is a superkey for relation R if $K \rightarrow R$, 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

4. Transaction Management

4.1 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 the 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 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)

4.2 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

4.3 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 (i.e., the element is 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 transactions 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 i (T_i) takes precedence over transaction j (T_j) 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 , A_1 and A_2 involve the same data element, and at least one is a write
 - Equivalently, T_i has precedence over T_j if there exists conflicting actions $A_1 \in T_i$ and $A_2 \in T_j$ such that A_1 precedes A_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

4.4 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

- ↳ This ensures that no other transactions may interfere with a given element until the transaction is completely finished with it
- To produce the schedule, each transaction can greedily request resources/locks, and the scheduler will either grant these or deny them

4.5 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

4.6 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
- There are five locking states of the database:
 - ↳ UNLOCKED. No locks are held on the database, and the database can not 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)
- ↳ It is possible that a write may occur sooner than the commit (perhaps when using the immediate/exclusive mode)
- ↳ 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

4.7 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
- Rollbacks can be handled

4.8 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
- 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 transaction 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)

4.9 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 transactions 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 not 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)

4.10 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

4.11 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

5. Relational Database Model

5.1 Structure of 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
 - ↳ The relation may be given a name
- 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

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
- SQL statements fall into two categories:
 - ↳ Data Definition Language (DDL), which build/modify the structure of the tables/objects, including **CREATE**, **ALTER**, and **DROP**
 - ↳ Data Manipulation Language (DML), including adding/removing/querying data
 - ↳ DML is often summarized by the acronym CRUD: create, read, update, and delete

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

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
 - ↳ 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)
- 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`

```

1 def adapt_color(color):
2     return f"{color.r};{color.g};{color.b}"
3 def convert_color(bytestring):
4     as_str = bytestring.decode("ascii")
5     r, g, b = [float(x) for x in as_str.split(';')]
6     return Color(r,g,b)
7 sqlite3.register_adapter(Color, adapt_color)
8 sqlite3.register_converter("COLOR", convert_color)
9 c = Color(24, 69, 59)
10 conn = sqlite3.connect(":memory:", detect_types = sqlite3.PARSE_DECLTYPES |
    ↳ sqlite3.PARSE_COLNAMES)
11 conn.execute("CREATE TABLE my_colors (col COLOR);")
12 conn.execute("INSERT INTO my_colors VALUES (?);", c)
13 res = conn.execute('SELECT col AS "col [COLOR]" FROM my_colors;')
14 row = next(res)

```

6.4 Adding Data

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

```

1 CREATE TABLE table_name (
2     column1 datatype,
3     column2 datatype,
4     column3 datatype
5 );

```

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

```

1 CREATE TABLE new_table AS
2 SELECT column1, column2
3 FROM old_table

```

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

```

1 INSERT INTO table_name (column1, column2, column3)
2 VALUES (value1a, value2a, value3a),
3         (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`:

```

1 INSERT INTO valued_customers
2 SELECT * FROM customers
3 ORDER BY money_spent DESC
4 LIMIT 20;

```

6.5 Updating Data

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

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

6.6 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

6.7 Querying

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

```
1 SELECT column1, column2, column3
2 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

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

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

↳ The order is ascending by default, although **DESC** can be specified to make it descending

- Rows can be filtered with the **WHERE** clause:

```
1 SELECT column1, column2
2 FROM table_name
```

- **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**
 - ↳ Since **DISTINCT** is an operator returning distinct values, it can also be used inside the functions described here, such as **count(DISTINCT col1)**
 - ↳ **DISTINCT** operates on a single column; multi-column **DISTINCT** is not supported
- Altogether, a **SELECT** statement may have the following parts, in order: **SELECT, FROM, WHERE, GROUP BY, HAVING, ORDER BY, LIMIT**

6.8 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\'**
- **IN** is used to select values that are in a list, such as **IN (1,3,5)** (or it can be used with subqueries)

6.9 SQL 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

6.10 SQL Aggregators

- Aggregators take 0 or more values and return some form of summary of those values
- **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

6.11 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)`

6.12 Case Expression

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

```

1 UPDATE table SET col =
2 CASE
3   WHEN cond_1 THEN val_1
4   WHEN cond_2 THEN val_2
5   ELSE val_3
6 END;
```

- A case expression can also use the value of some expression instead of conditions:

```

1 UPDATE table SET col =
2 CASE expr
3   WHEN expr_val_1 THEN val_1
4   WHEN expr_val_2 THEN val_2
5   ELSE val_3
6 END;
```

6.13 Column Constraints

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

```

1 CREATE TABLE students (
2     msu_id TEXT UNIQUE,
3     pid INTEGER UNIQUE,
4     first_name TEXT)
5 );

```

↳ 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)

6.14 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:

```

1 CHECK (col_in_table IS NULL OR EXISTS (
2     SELECT 1 FROM table
3     WHERE table.id = col_in_table.id
4 ))

```

6.15 Qualified Table Names

- 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**
- Implicit aliases are not recommended but possible, 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

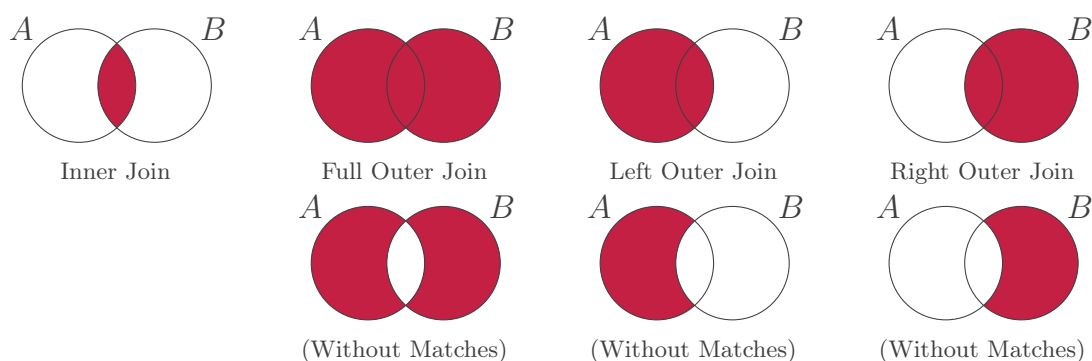
6.16 SQL Union

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

```
1 SELECT Student.Name FROM Student
2 UNION
3 SELECT Staff.Name FROM Staff;
```

- The columns need not have the same name; the only requirement is that the datatypes are the same
- UNION** removes duplicates on the results; to avoid this, use **UNION ALL**

6.17 SQL Joins



- 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 joins 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 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 followed 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:

```
1 SELECT * FROM tableA
2 LEFT OUTER JOIN tableB
3 ON tableA.col = tableB.col
4 UNION ALL
5 SELECT * FROM tableB
6 LEFT OUTER JOIN tableA
7 ON tableB.col = tableA.col
8 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:

```

1 SELECT * FROM table AS t1
2 INNER JOIN table as t2
3 ON t1.column1 = t2.column2
4 AND t1.column2 = t2.column1
5 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

6.18 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
- 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:

```

1 -- Select artists who have not released an album
2 SELECT Artist.Name FROM Artist
3 WHERE Artist.Name NOT IN
4 (SELECT Album.ArtistId FROM Album);

```

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

```

1 -- Select courses offered in both fall and spring
2 SELECT name FROM courses
3 WHERE semester = 'Fall'
4 AND name IN
5 (SELECT name FROM courses
6 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:

```

1 -- Select suppliers who make clothing
2 SELECT SupplierName FROM Suppliers
3 WHERE EXISTS (
4     SELECT ProductName FROM Products
5     WHERE Products.SupplierID = Supplier.SupplierID
6     AND Category = 'Clothing'
7 );

```

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

- **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)`

```

1  -- Select customers who have paid for a track of over 5 dollars
2  SELECT CustomerId FROM Invoice
3  WHERE InvoiceId = ANY(
4      SELECT InvoiceId FROM InvoiceLine
5      WHERE UnitPrice>5
6  );

```

↳ 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**

6.19 Group By

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

```

1  -- Count how many tracks of each genre are present
2  SELECT GenreId, count()
3  FROM Track
4  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

```

1  -- Count how many tracks of each genre are present, but only if they have at
   ↳ least 10 tracks
2  SELECT GenreId, count()
3  FROM Track
4  GROUP BY GenreId
5  HAVING count()>10;

```

6.20 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

6.21 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:

```

1 DELIMITER //
2 CREATE PROCEDURE add_now_to_log()
3 BEGIN
4     INSERT INTO log VALUES (datetime('now'));
5 END //
6 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

```

1 DELIMITER //
2 CREATE PROCEDURE increment (INOUT tally INT(4), IN inc INT(4))
3 BEGIN
4   SET tally = tally + inc;
5 END //
6 DELIMITER ;
7 SET @count = 0;
8 CALL increment(@count, 3);
9 SELECT @count;
```

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

```

1 CASE
2   WHEN cond_1 THEN statement_1;
3   WHEN cond_2 THEN statement_2;
4   ELSE statement_3;
5 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 trace-back and error reporting
 - ↳ Difficult to maintain: Stored procedures are less transparent and harder to write, especially if the schema changes

6.22 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:

```

1 CREATE TRIGGER trigger_name [timing] event_name
2 ON table_name [WHEN predicate] BEGIN
3   commands;
4 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.***

```
1 CREATE TRIGGER id_change AFTER UPDATE ON students WHEN NEW.id != OLD.id
2 BEGIN
3     UPDATE log SET student_id = NEW.id WHERE student_id = OLD.id;
4 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
- 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

6.23 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
- Similar drop statements exist for triggers with **DROP TRIGGER**

6.24 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**

6.25 Indices

- Most ordinary tables 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**
- 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

6.26 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)

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

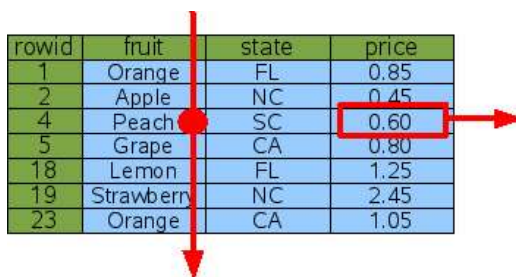
Figure 1: Logical Layout Of Table "FruitsForSale"

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 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

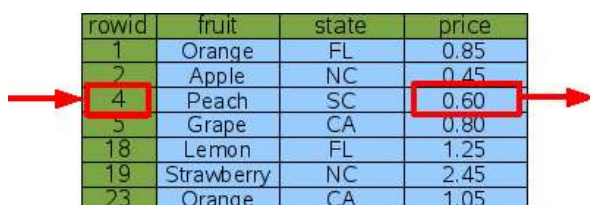
Figure 2: Full Table Scan

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 $\log N$ 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 $N/\log N$ 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 engine 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](#).

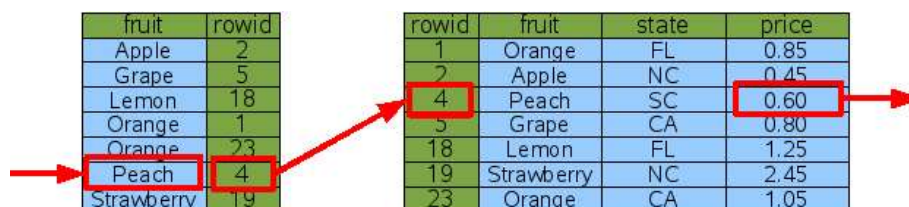


Figure 5: Indexed Lookup For The Price Of Peaches

SQLite has to do two binary searches to find the price of peaches using the method shown 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';
```

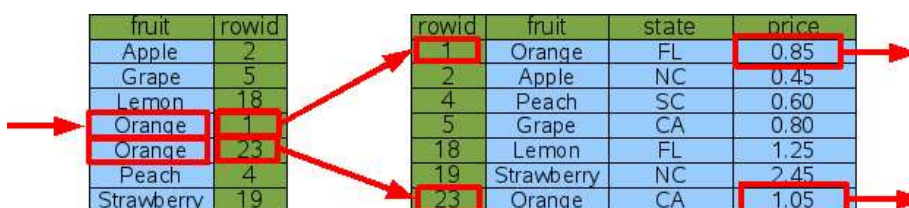


Figure 6: Indexed Lookup For The Price Of Oranges

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 $(K+1) \cdot \log N$.

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';
```



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.



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'
```

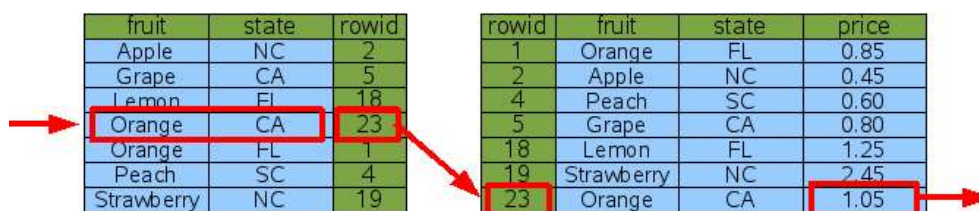


Figure 11: Lookup Using A Two-Column Index

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'
```

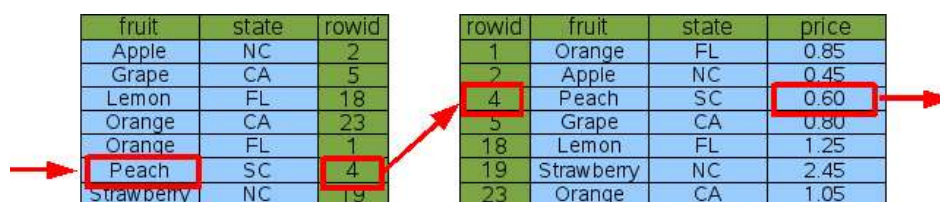


Figure 12: Single-Column Lookup On A Multi-Column Index

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:

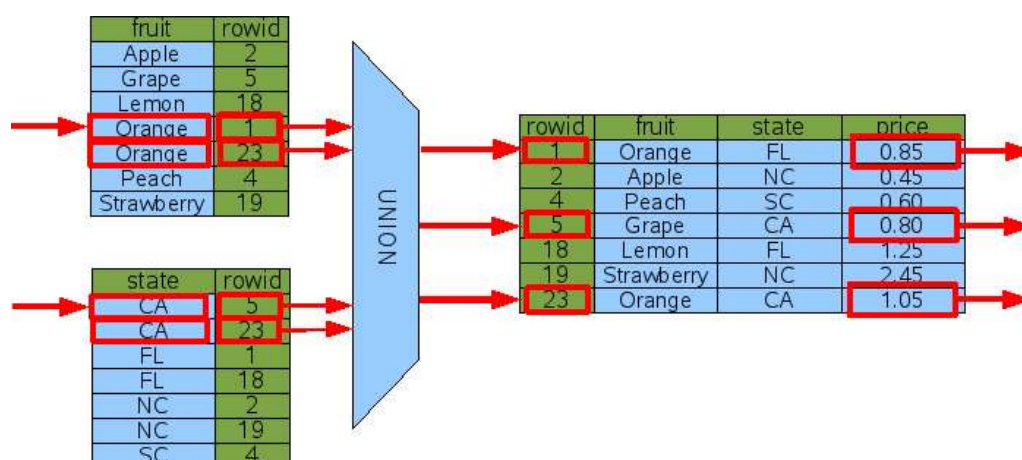


Figure 15: Query With OR Constraints

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.



Figure 16: Sorting Without An Index

If the number of output rows is K , then the time needed to sort is proportional to $K \log K$. 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;
```

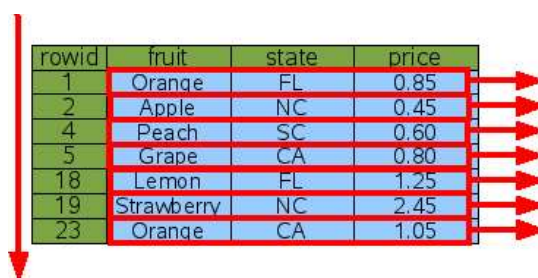


Figure 17: Sorting By Rowid

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;
```

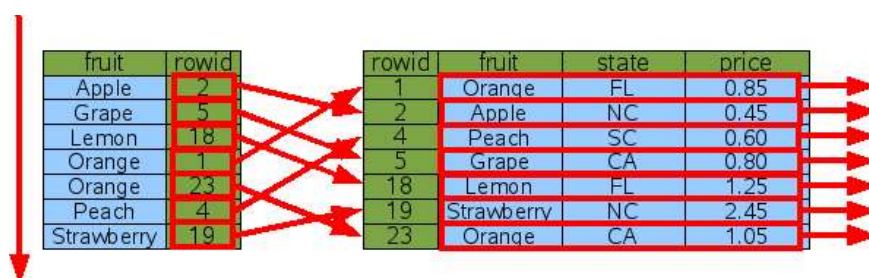


Figure 18: Sorting With An Index

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 $N \log N$ 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 $\log N$ time each, so the total time of $N \log N$ 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.



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.

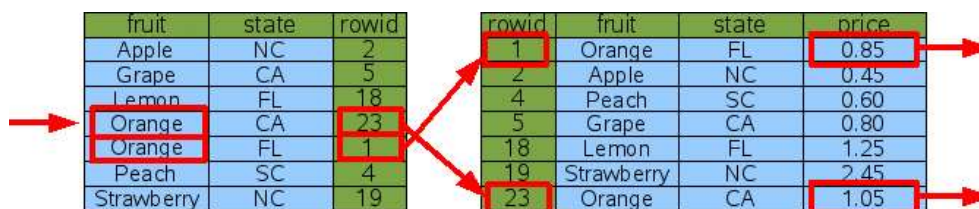


Figure 20: Search And Sort By Multi-Column Index

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
```

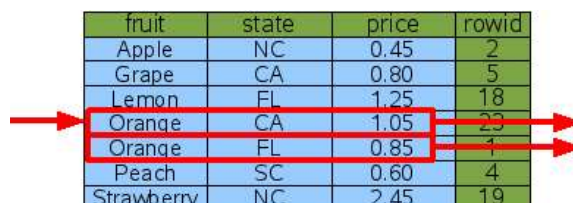


Figure 21: Search And Sort By Covering Index

As before, SQLite does single binary search for the range of rows in the covering index that satisfy the WHERE clause, then 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.



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](#)