File Structures and Indexing

CSCI 220: Database Management and Systems Design

Slides adapted from
Simon Miner
Gordon College

Practice Quiz: 4NF

- Identify the multivalued dependency
- Normalize the schema to 4NF

Book

ISBN	title	publisher	author
0133970779	Fundamentals of Database Systems	Pearson	Ramez Elmasri
0133970779	Fundamentals of Database Systems	Pearson	Shamkant Navathe
1934356557	SQL Antipatterns	pragprog	Bill Karwin
0393866661	A Hacker's Mind	Norton	Bruce Schneier

Agenda

- Mid-Semester Feedback
- Database Hardware
- Database File Structures
- Indexing

Today you will learn…

- How computer hardware affects database performance
- How databases organize information on disk

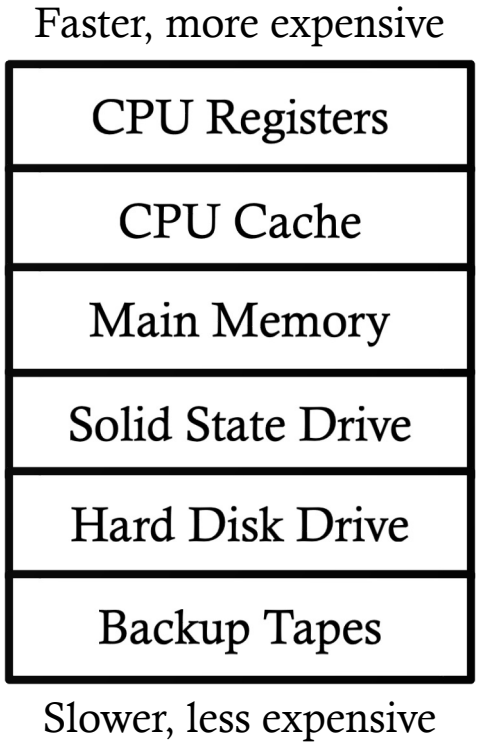
Database Hardware

File System Performance

- Often *the* major factor in DBMS performance
  - **Minimize response time:** time between issuing a command and seeing its results
  - **Maximize throughput:** number of operations per unit of time
    - Especially important for a system with many users (i.e. large scale web site)

Physical Storage

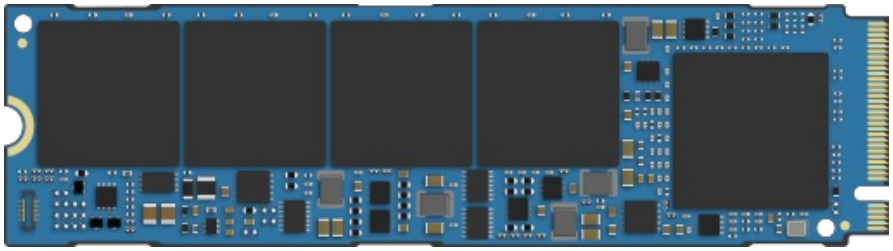
- Primary storage (memory)
  - Fast
  - Volatile: lost on power failure
- Secondary storage (disk)
  - Slower (online storage)
    - 100,000x slower than memory
  - Non-volatile
- Tertiary storage (tape)
  - Not immediately available (offline storage)



Secondary Storage: SSD vs HDD Performance

- SSDs faster all around, because they don't have physically moving parts
- However, HDDs have lower price per TB

William Warby, Wikimedia Commons



Tom Cowap, Wikimedia Commons

SSD vs HDD Cost

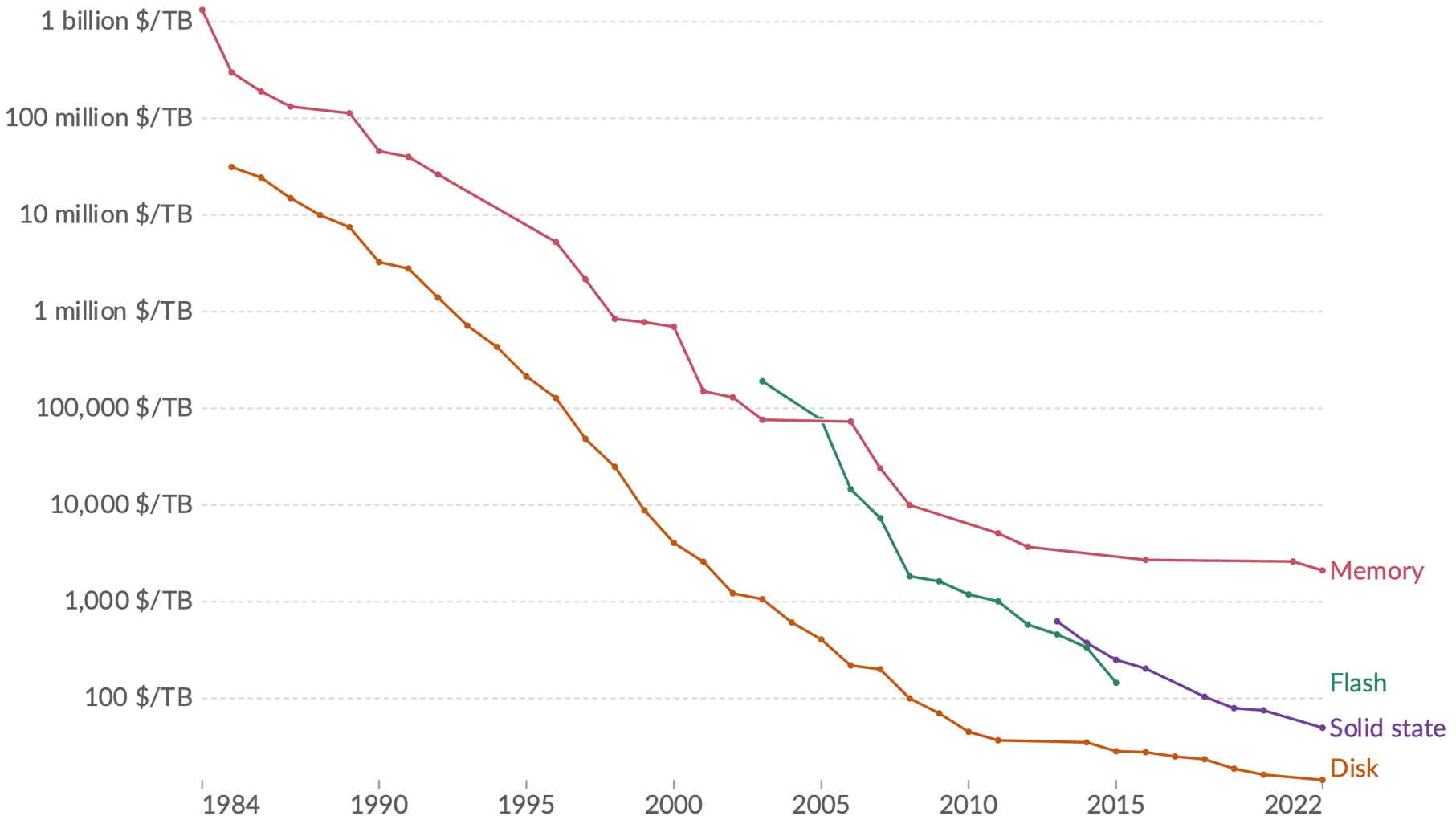
- Cheapest HDD (by price/TB):
  - Seagate Enterprise Capacity 3.5 HDD 6TB 7200RPM 12Gb/s SAS
  - Capacity: 6TB
  - Price: $65, **($10.83/TB)**
- Cheapest SSD (by price/TB):
  - Verbatim Vi7000 NVMe PCIe M.2 2280 Internal Gaming SSD
  - Capacity: 2TB
  - Price: $94 **($46.94/TB)**

https://diskprices.com/

Historical cost of computer memory and storage

This data is expressed in US dollars per terabyte (TB). It is not adjusted for inflation.

Our World in Data



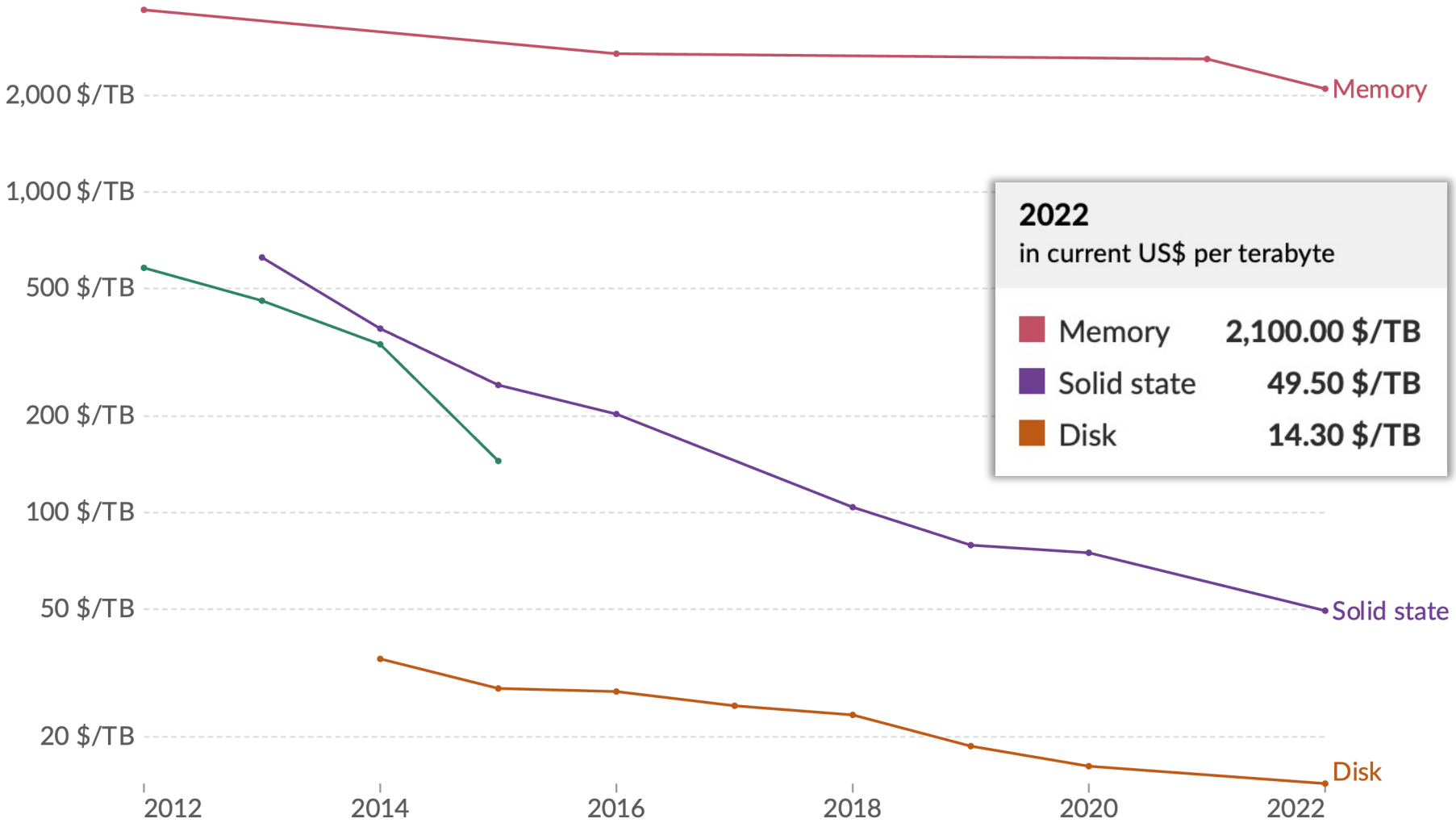
Data source: John C. McCallum (2022)

OurWorldInData.org/technological-change | CC BY

Note: For each year, the time series shows the cheapest historical price recorded until that year.

Historical cost of computer memory and storage

This data is expressed in US dollars per terabyte (TB). It is not adjusted for inflation.



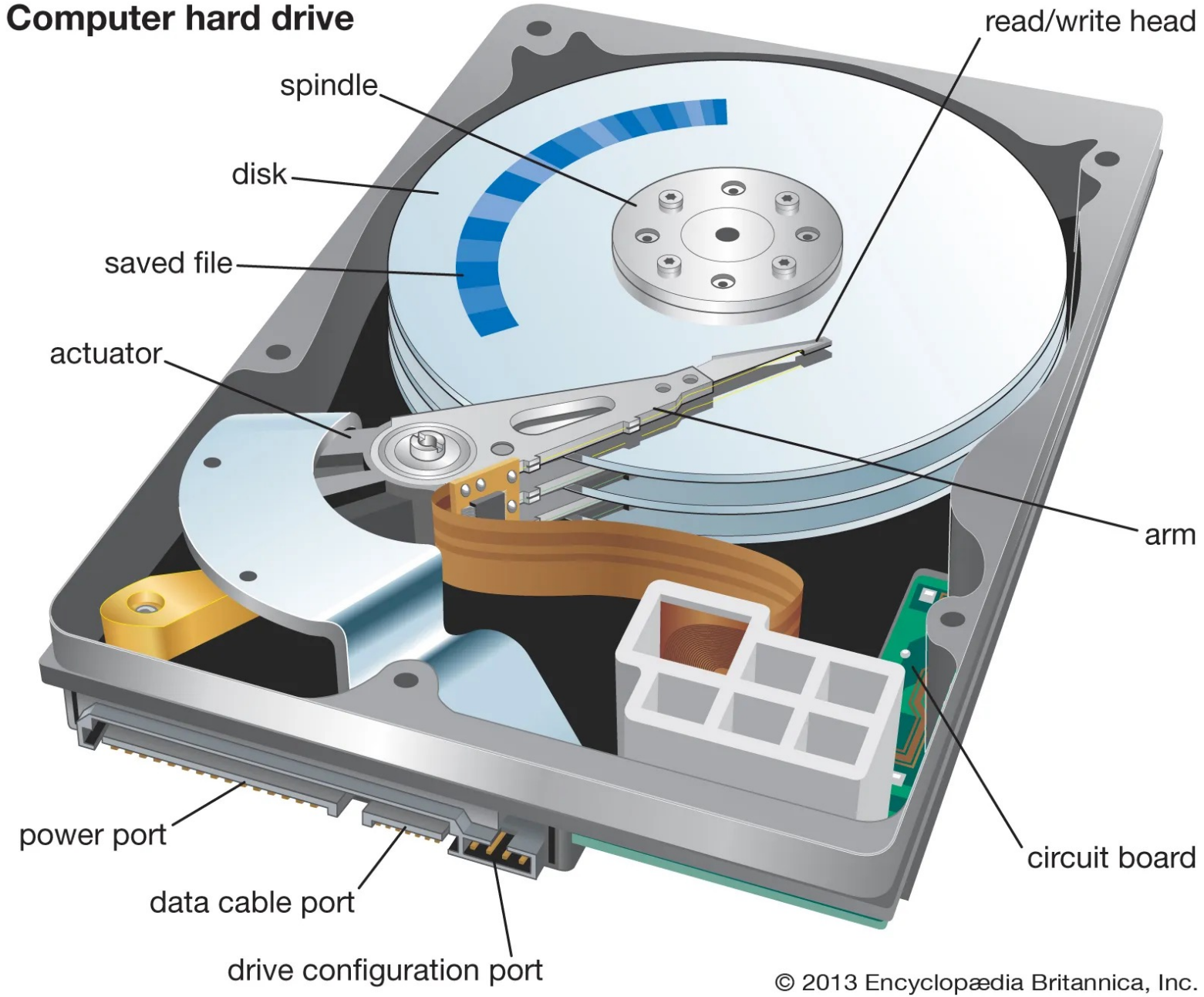
Data source: John C. McCallum (2022)

OurWorldInData.org/technological-change | CC BY

Note: For each year, the time series shows the cheapest historical price recorded until that year.

Other Cost Factors

- Energy use and cooling: SSD wins
- Average time-to-failure: depends
- Number of units needed: depends
  - Do you care about total storage, or total throughput?


Computer hard drive
read/write head
spindle
disk
saved file
actuator
arm
power port
data cable port
drive configuration port
circuit board
© 2013 Encyclopædia Britannica, Inc.

HDD Access Time

- How long it takes to read or write data to disk. Includes:
  - Seek time: time needed to to position the disk head to the correct track (4-10 ms)
  - Rotation latency: time needed to rotate the disk so that the desired information starts to pass under the head (4-11 ms for typical disks 5400 – 15000 rpm)
  - Data-transfer rate: time needed to transfer information
    - ~1% of total time
- To optimize this process, data on disk is organized into blocks
  - Chunks of contiguous information
  - System reads or writes entire blocks, not individual bytes

How a DBMS Minimizes Disk Access

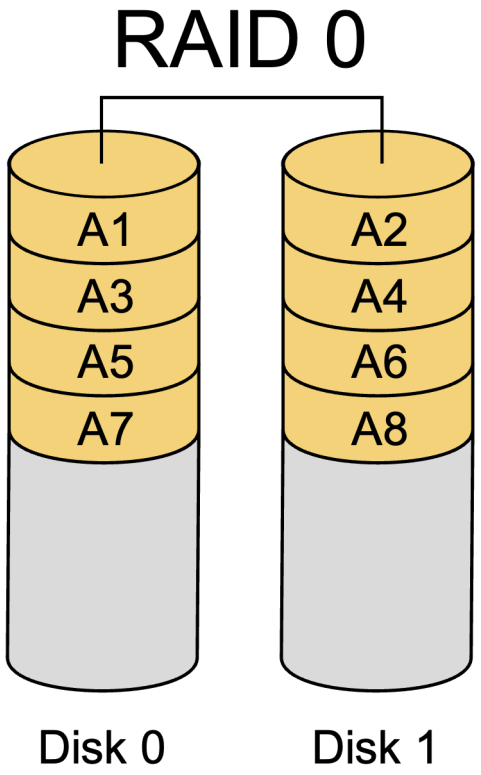
- Keep all data needed for a particular operation in a single block
  - Only one disk access needed
- Keeping copies of recently used information in memory
  - Disk access needed for initial operation, but repeated or similar operations can use in-memory copy
- Parallelism: spread data across multiple disks
  - Data access happens on several disks at the same time

RAID: Redundant Arrays of Independent Disks

- RAID manages multiple disks, and provides a view of a single disk
- Pick two: speed, redundancy, capacity
- Protection against hardware failure is essential!
  - The chance that a single disk will fail in a large system increases as the number of disks goes up
  - e.g., a system with 100 disks, each with MTTF of 100,000 hours (approx. 11 years), will have a system MTTF of 1000 hours (approx. 41 days)

RAID 0

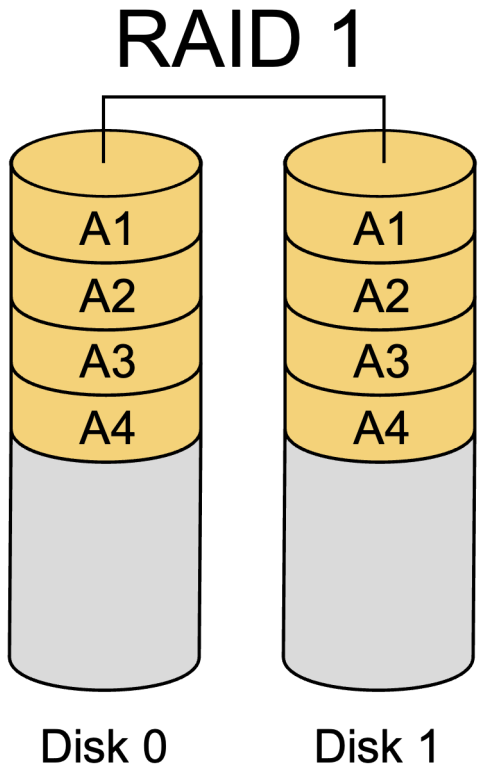
- Striping: spread data across multiple disks
- Pros:
  - Fast reads and writes
  - Fully utilizes storage space
- Cons:
  - **A single disk failure will corrupt all data**



Credit: Cburnett

RAID 1

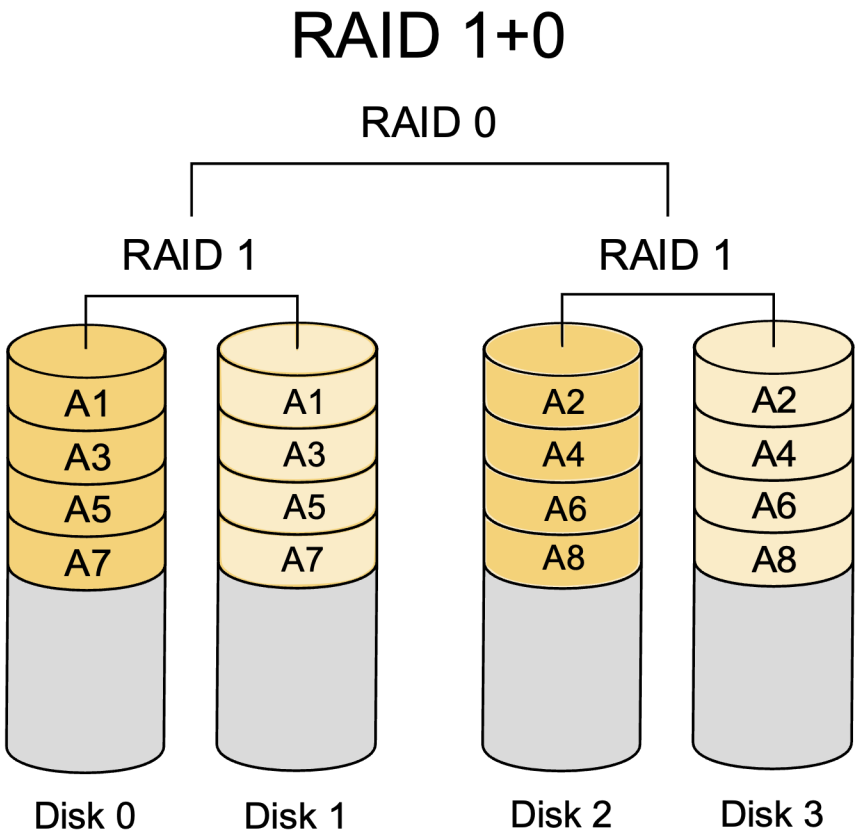
- Mirroring: copy data onto multiple disks
- Pros:
  - Fast reads
  - High reliability
- Cons:
  - Slow writes
  - Inefficient use of storage



Credit: Cburnett

RAID 10

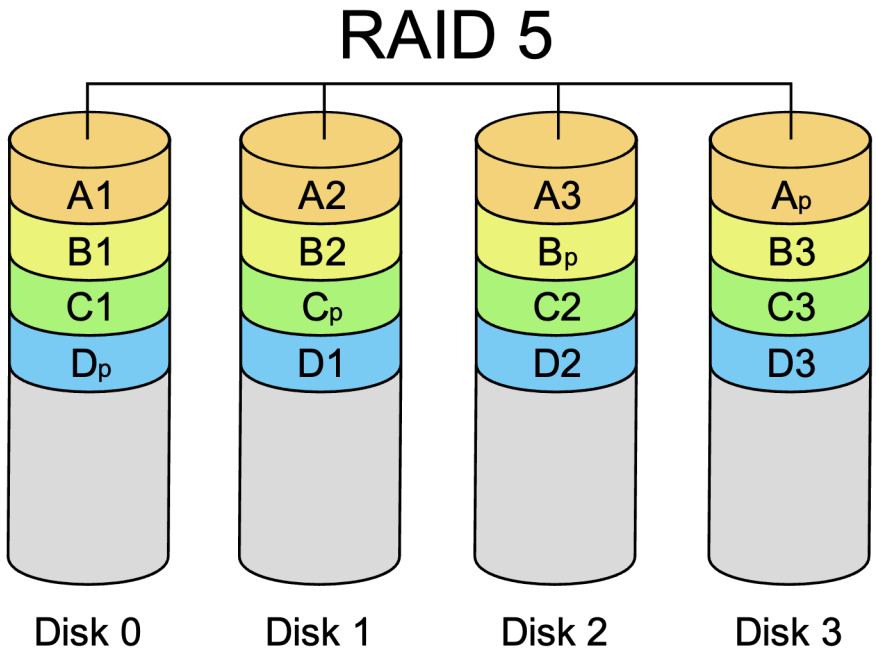
- A stripe of mirrors (order matters!)
- Pros:
  - Fast reads and writes
  - High reliability
- Cons:
  - Inefficient use of storage



Credit: Cburnett

RAID 5

- Combines striping with parity data
- Pros:
  - Fast reads
  - High reliability
  - Efficient use of storage
- Cons:
  - Slow writes



Credit: Cburnett

Database File Structures

What data is stored on disk?

- User data
- System catalog
- User access control data
- Statistical data
- Index data
- Logging data

How is data stored on disk?

- Data is stored as one or more files
  - As a collection of files (e.g., MySQL uses one file for each table)
  - A single large file on the operating system within which the DBMS builds its own file system (e.g., DB2)
  - Hybrid of these approaches (e.g., Oracle)

How is data accessed?

- Data is accessed through the "buffer manager," which manages a memory pool (i.e., a buffer)
  - Stores most recently accessed block from disk for each table (at a minimum)
  - Often, retains data that has been used once and is likely to be used again
  - Logic needed to manage what data is kept in the pool
    - Since memory is usually smaller than the entire database

Other Software Components

- Query parser: accepts and translates queries
- Strategy selector: plans the best way to carry out queries
  - Uses statistical data on table sizes, indices, etc.
- Crash recovery manager: restores data to a consistent state after a failure
  - Uses a log of changes made to the database
- Concurrency controller: prevents inconsistencies from simultaneous changes by multiple users

File Organization Approaches

- Fixed-length records
- Variable-length records

Fixed-Length Records

- Every record is allocated the same amount of space
  - Records of the same type can reside in a single file (or portion of a file
  - Record offset = (relative position – 1) * record size
- Space from deleted records can be reused
  - Move all records after the deleted one back one slot (expensive)
  - Move the last record into the empty slot (less expensive)
  - Link free slots together in a *free list*
    - Address of first free (deleted) stored in file header
    - Each deleted record stores the address of the next deleted record

Variable-Length Records

- Fixed-width records are not always practical
  - Storing arbitrarily long pieces of text (i.e. articles, documents)
  - Storing binary resources (i.e. pictures, videos)
- Approaches
  - Represent variable-length attributes with a fixed size (offset, length), and store their actual values after all other data in the record
  - Store fixed-length record data in one file with pointers to variable-length data in other files
    - Multimedia databases may have pointers to individual files for variable-length values (“clobs” and “blobs”)

Record Organization

- Sequential: sort records in a table by some column value
  - Good choice if most/all queries of the table are done using the sorted criteria
  - Inserts become problematic – need to retain sort order
    - "Buckets" can be used to help address this – all records with same or similar sort key values go into the same bucket
      - Reduces cost of preserving sort order
- Multi-table clustering: sometimes data in multiple tables is related and queried together
  - Store related data from each table on the same (set of) disk block(s)
  - Good for queries involving related data, not so go for queries on individual records
  - Results in variable-length records
  - NoSQL solutions often use this approach

Buffer Management

- How does the DBMS decide which data is tossed from the buffer when new query results are being loaded?
- Policies
  - Least Recently Used (LRU) – toss the buffer contents which have not been used for the longest period of time
    - Based on the idea that past query patterns are a good predictor of the future
  - Most Recently Used (MRU) – toss the buffer contents which have been used most recently
    - Good when cycling through contents of a table which is too big for memory
  - Based on frequency of block usage
    - Examples: blocks in the data dictionary, root blocks of indexes

Indexing

Indexing Overview

- Indexes (indices) used to efficiently search for row(s) in a table that match certain criteria
  - Find the disk block with the desired data with minimal disk accesses
- Index trade-off
  - Improved search efficiency vs cost of maintaining the index
    - Disk space required for index, and time needed to maintain it
- Search key
  - Attribute(s) used to do lookups on an index
  - Multiple indexes can be created on a table with different search keys

Index Considerations

- What will the index be used for?
  - Find rows which match exact values
  - Range queries (i.e. values between, greater, or less than some bounds)
  - Sequential access of all rows in the table
- How frequently is the underlying data modified?
  - Lots of inserts, updates, and deletes mean more index maintenance
  - Read-only / read-mostly data can use indexes that facilitate faster data access but are expensive to maintain
- Is the search key a superkey (or the primary key)?
- Can multiple rows share a single key value?

Ordered vs Hashed Indexes

- Ordered indexes keep index entries in the order of the search key
  - Facilitates range queries and accessing all rows in search key order
  - Typically structured as B+ trees
- Hashed indexes use a hashing function to evenly distribute index entries among blocks
  - Offers more efficient access and maintenance
  - Doesn't support ordered operations

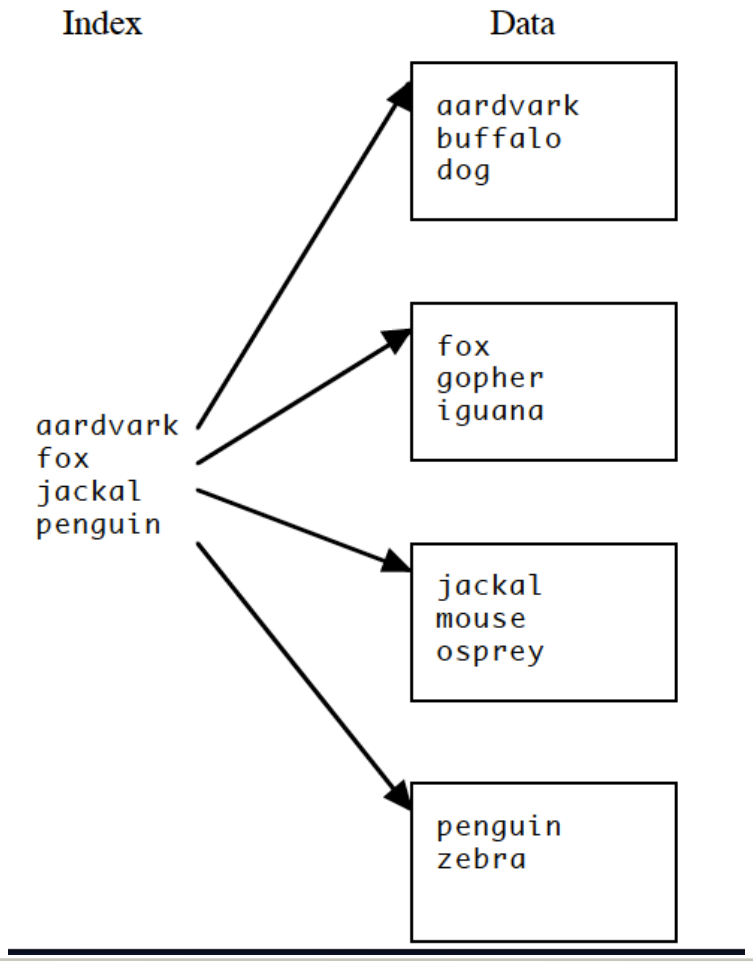
Clustered Index

- Actual data is stored in the order dictated by the index
  - Only applies to ordered indexes
  - A given file can have at most one clustering index
- Advantages
  - Makes range queries easier – only need to locate first row in the range, and then read subsequent rows
  - Makes accessing rows with the same search key value easier, as they will be adjacent
- Disadvantage
  - Hard to maintain – inserts, updates, and deletes all require moving data
- Sometimes called a primary index (or an index organized table)
  - Other indexes can be referred to as non-clustering or secondary

Dense vs Sparse Indexes

- Dense index has one index entry for each distinct search key value
- Sparse index does not
  - Only a clustering index can be sparse – index is used to locate the starting point for a search of the actual data
    - Using the largest entry <= desired value
  - Sparse index typically contains one entry for each data block in the file (the smallest search key value in the block)

Sparse Index Example



Multilevel Index

- If primary index does not fit in memory, access becomes expensive
- Solution: treat primary index kept on disk as a sequential file and construct a sparse index on it
  - outer index: a sparse index of primary index
  - inner index: the primary index file
- If even outer index is too large to fit in main memory, yet another level of index can be created, and so on
- Indices at all levels must be updated on insertion or deletion from the file

Indexing, Continued

Practice Quiz: RAID

With a neighbor, discuss which RAID level is most appropriate for these scenarios:

- My lab captures hour-long video recordings of interviews with study participants
  - Priorities: redundancy, storage capacity
- A credit card company processes transactions from millions of customers each day
  - Priorities: redundancy, read and write speed
- A professional gamer wants games to load as fast as possible
  - Priorities: read speed, redundancy

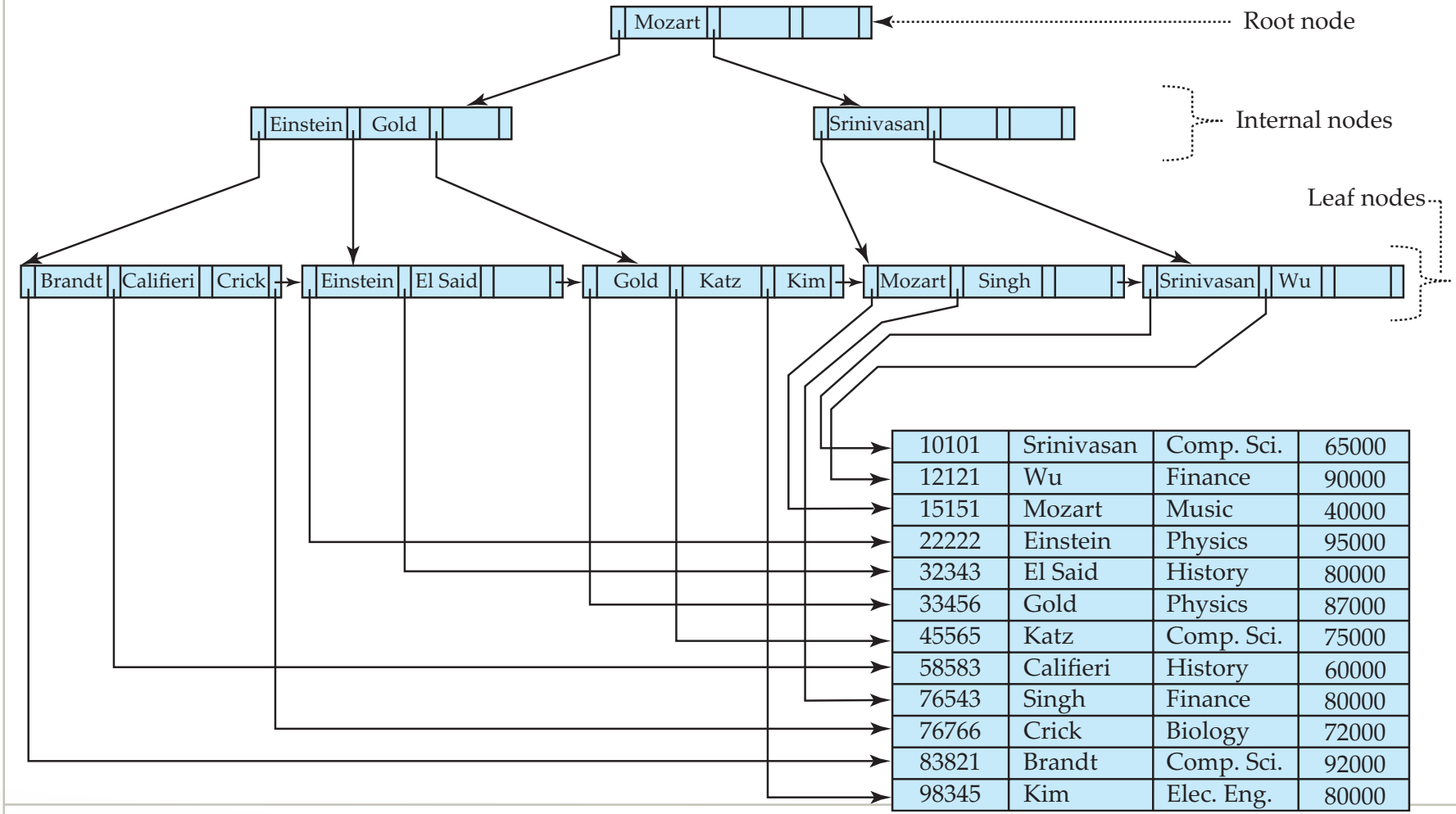
Agenda

- Indexing
- Database Design Tips

B+ Tree Indexes

- Alternative to clustered indexes
- May be used both primary and secondary indexes
  - As primary index, the tree can contain both index data and the actual records in the table
  - As secondary index, the tree only contains index data
- Advantage of B^+-tree index files:
  - Automatically reorganizes itself with small local changes when inserts, updates, and deletes occur
  - Reorganization of entire file is not required to maintain performance

B+ Tree Example



10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	80000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	60000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

B+ Tree Structure

- Multi-leveled with all leaf nodes on the same level
- The *order* (n) of a B+ tree determines the maximum number of keys per node
  - Constrained by the size of a node and the size of a key-value pair
- Components
  - Root (with at least 2 children)
  - Internal (non-leaf) nodes
  - Leaf nodes

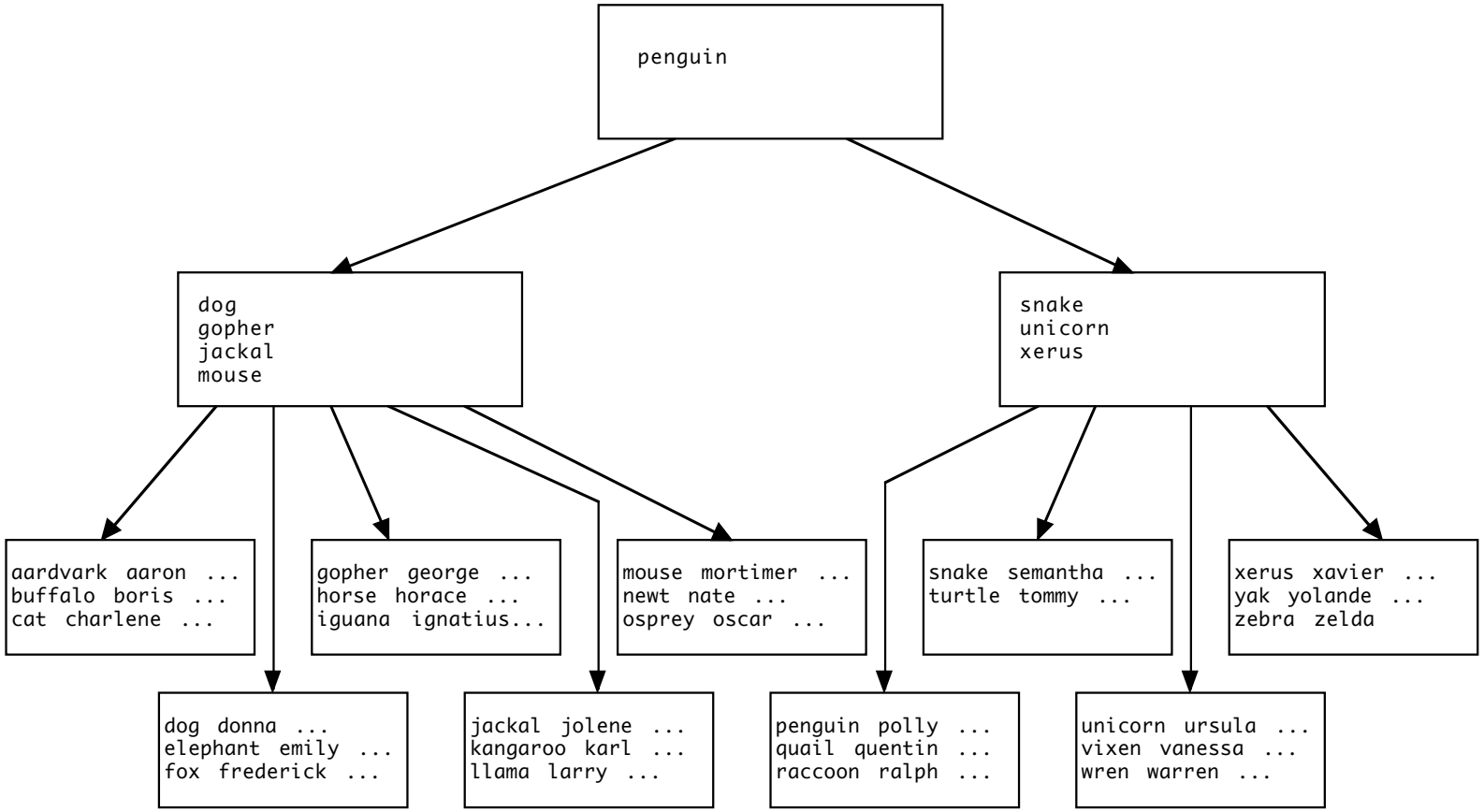
Internal (Non-leaf) Nodes

- Contain index data: pairs of search key values and pointers to other nodes on the next level of the tree
- Can hold between $\lceil (n-1)/2 \rceil$ and n-1 keys
- Has between $\lceil n/2 \rceil$ and n children
  - A node with k keys has k+1 children
  - Key values separate pointers to nodes or records on next level
- Keys in a node are ordered

Leaf Nodes

- Comprised of one of the following
  - Index data: pairs of search key values with pointers to actual records
    - Contain between $\lceil (n-1)/2 \rceil$ and n-1 search key values
    - Last pointer in an indexing leaf node points to next leaf node (instead of a record)
  - Actual records
    - In primary index
    - Number of records in a leaf depends on the size of each record (separate from order of the B+ tree)
    - May also include pointer to next leaf

An Order 7 B+ Tree

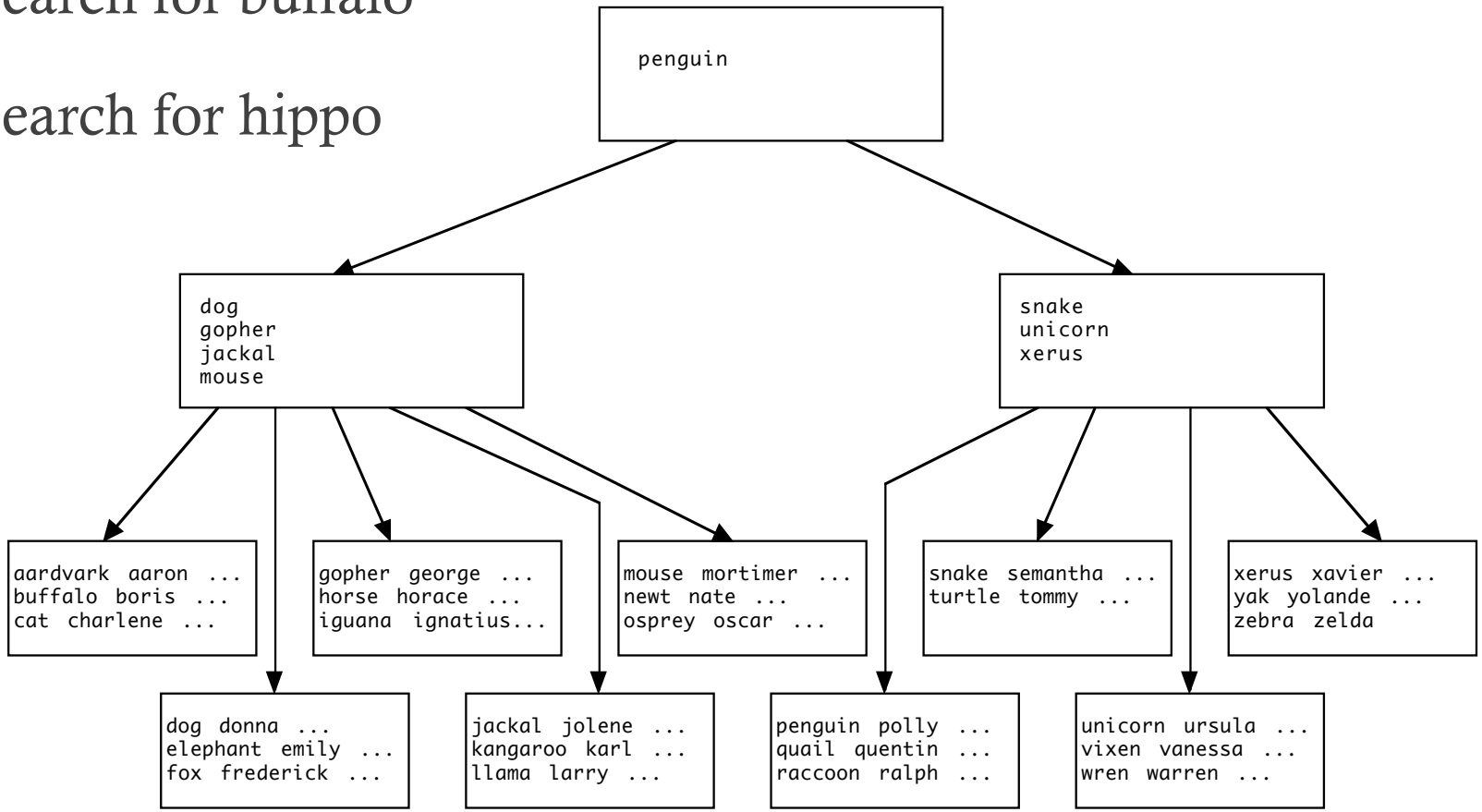


Searching the B+ Tree

- Algorithm
  - Start at the root
  - while at an internal node:
    - if the value being sought is less than the smallest key stored in the node
      - go to the leftmost child
    - else
      - go to the child corresponding to the largest stored key that is ≤ the value
        - Note: the second child corresponds to the first key
  - When we reach a leaf node, the desired value will either
    - Be contained in the leaf (found by a sequential search within the node)
    - Not exist in the tree

B+ Tree Search Examples

- Search for buffalo
- Search for hippo

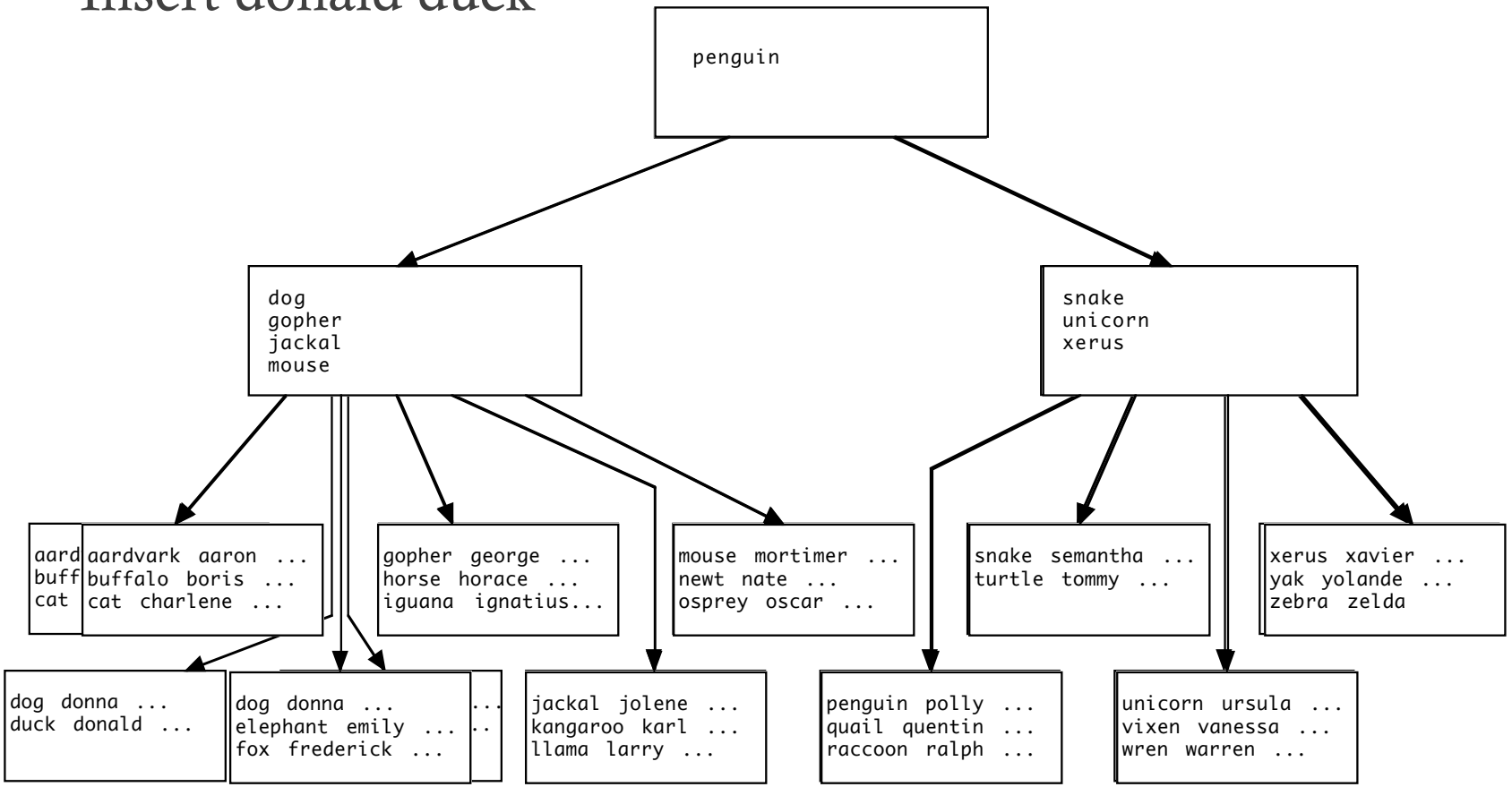


Inserting into the B+ Tree

- Algorithm
  - Use search procedure to find node where it would be if it was present.
  - if there is room,
    - put it there.
  - else
    - divide the keys in two
    - create a new right block to contain half the keys
    - "promote" the first key in the right block. Insert this key, plus a pointer to the new right block, in the parent
- This may cause the parent to split
  - In this case, create a new internal node and promote the "split key" to the parent
- Root may actually split as well
  - Create new root with halves of original root as its children

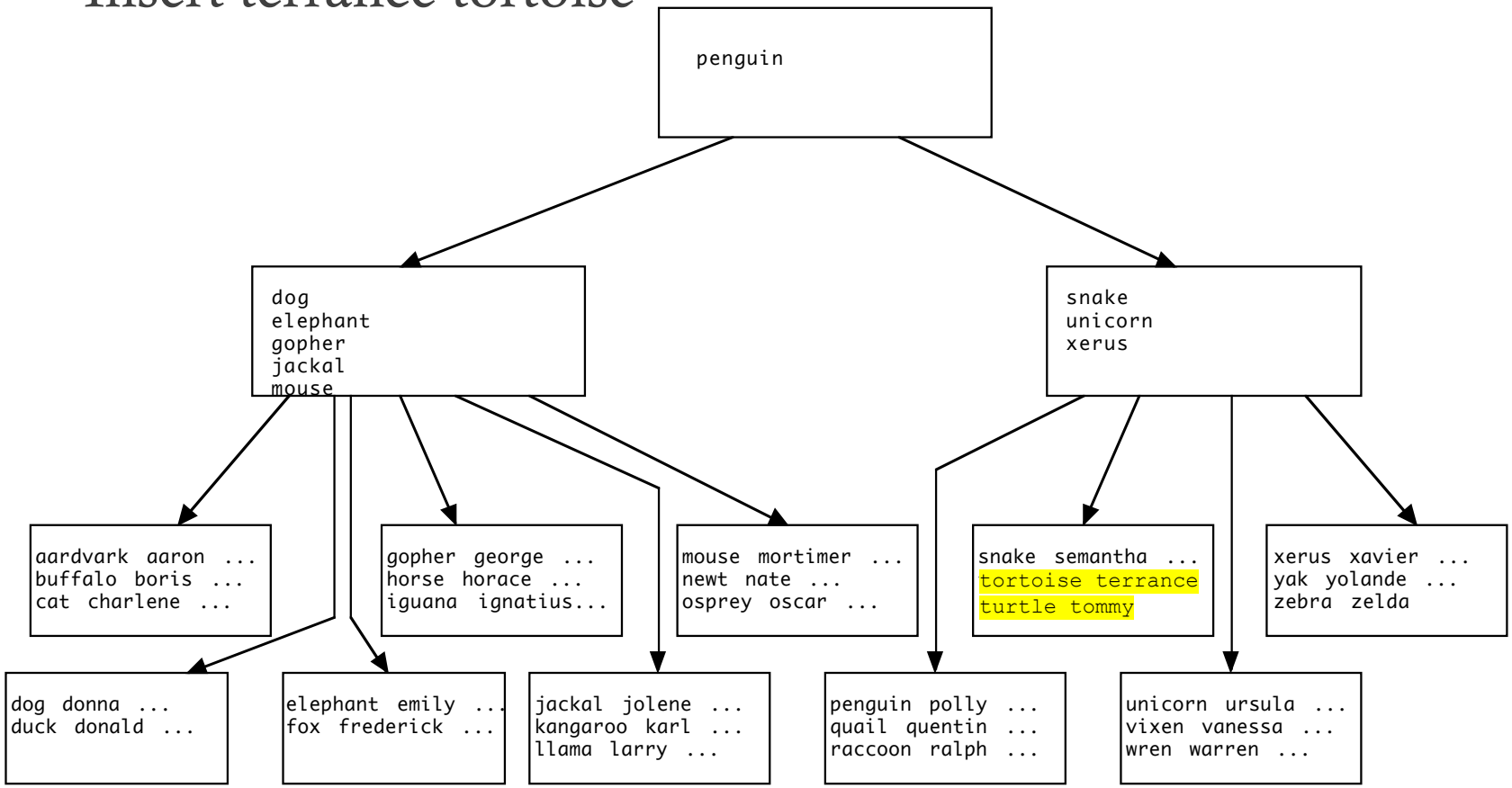
B+ Tree Insert Example

- Insert donald duck



B+ Tree Insert Example

- Insert terrance tortoise



Hashing

- Alternate index structure facilitating fast access
  - Search key hashed to look up records (primary index)
  - Search key hashed to look up record pointers (secondary index)
- Records (or record pointers) reside in one of several buckets
  - A hashing function on the search key determines which bucket a record/pointer goes in

Hashing Functions

- Worst hash function maps all search-keys to the same bucket
  - Makes access time proportional to the total number of search-keys
- An ideal hash function is **uniform**
  - i.e., each bucket is assigned the same number of search-keys from the set of *all* possible search-keys
- And ideal hash function is **random**
  - i.e., each bucket will have the same number of search-keys assigned to it irrespective of the *actual distribution* of search-keys
- Typically, hash functions perform computation on the internal binary representation of the search-key
  - For example, for a string search-key, the binary representations of all the characters in the string could be added and the sum modulo the number of buckets returned

Hashing Example

- Hashing function*
  - Sum of ASCII codes in first name
  - Modulo 7
- After locating a bucket, perform a sequential scan of the bucket
- Examples:
  - Look up tommy (bucket 6)
  - Look up terrance (bucket 5)

Bucket	Keys	Values
0		
1	larry	llama
2	yolanda quentin	yak quail
3	donna zelda	dog zebra
4	nate aaron mortimer	newt aardvark mouse
5	samantha emily	snake elephant
6	tommy	tortoise

Hashing Challenges

- What happens when a bucket runs out of room?
  - Because of an insufficient number of buckets
  - Because multiple records have the same search key (and hence, hash value)
  - Because the hashing function is non-uniform
- Possibilities
  - Overflow buckets
  - Reorganize the file with a new hashing function
  - Extendable hashing: dynamically modify the number of buckets

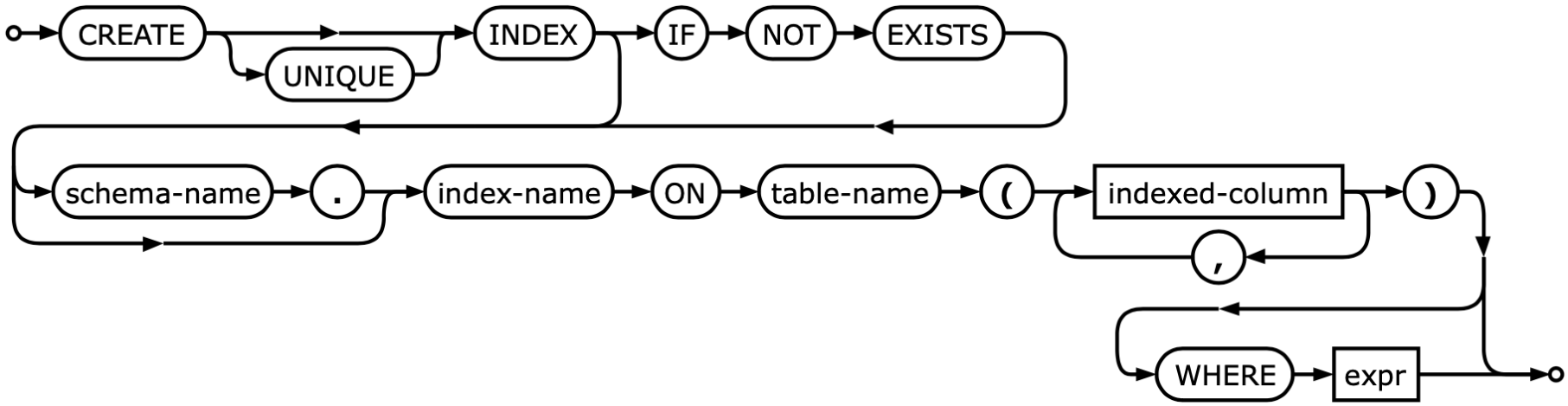
Comparison of Ordered and Hashed Indexes

- Hashed indexes
  - Allow fast access for exact match queries: usually 1 or 2 disk accesses (for primary and secondary indexes, respectively)
  - Do not support range queries or sequential access of entire table
- Ordered Indexes
  - Slower access: potentially several disk accesses as you work through the B+ tree levels
  - Supports more types of queries

Creating Indexes

- Database automatically creates indexes for:
  - Primary keys
  - Columns with unique constraints
  - (Sometimes) temporary indexes used for single queries

SQLite Create Index



https://www.sqlite.org/lang_createindex.html

PostgreSQL Create Index

```
CREATE [ UNIQUE ] INDEX [ CONCURRENTLY ] [ name ] ON table [ USING method ]

    ( { column | ( expression ) } [ COLLATE collation ]
        [ opclass ] [ ASC | DESC ] [ NULLS { FIRST | LAST } ] [, ...] )

    [ WITH ( storage_parameter = value [, ... ] ) ]

    [ TABLESPACE tablespace ]

    [ WHERE predicate ]
```

https://www.postgresql.org/docs/9.1/sql-createindex.html

Database Design Tips

What to Index

- Primary key (automatic)
- Columns with unique constraints (automatic)
- Foreign key columns
- Fixed-width columns: Boolean, numeric, (fixed-width) character, date/time fields
- Columns appearing in a “where” clause
  - Especially a “where” clause in a program (likely to be executed multiple times)
  - Including variable-width character fields
- Note: Indexes on NULL-able attributes may have unexpected behavior
  - e.g., indexes might not accelerate queries for NULL values
  - Research your DBMS's behavior

Don’t Index…

- Small tables (< 100 records) that will stay small
  - i.e. list of states and their capitals
- Columns containing binary (blob) or large text (clob) data
- Long-ish variable width text fields (i.e. product descriptions, review text, comments)
- Columns containing data that may be fetched or updated, but will never appear in a "where” clause

Index Names

- Unless you're using a framework, explicitly name your indexes
  - Don’t let the database make up names for you
- Index naming conventions
  - Begin with 2-5 letter prefix of table name or abbreviation
  - Column name(s) or abbreviation(s) that comprise the index’s search key
  - End with a date stamp (i.e. 20121011)
    - Gives the index a unique name
    - Helpful when you need to rebuild the index or copy the table
  - Prefix or suffix the following indexes
    - Primary keys: “pk”
    - Foreign keys: “fk”
    - Unique constraints: “uniq”