Storage and Efficiency

G51DBS Database Systems

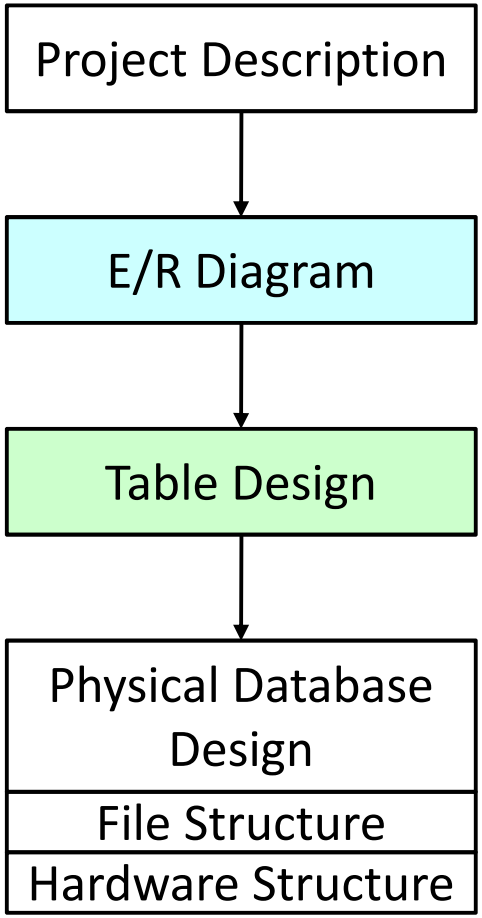
Jason Atkin

This Lecture

- Physical Database Design
  - RAID Arrays
  - Parity
- Database File Structures
- Indexes
- Further reading
  - The Manga Guide to Databases, Chapter 5
  - Database Systems, Chapters 7, 18

Physical Design

- Design so far
  - E/R modelling helps find the requirements of a database
  - Normalisation helps to refine a design by removing data redundancy
- Next we need to think about how the files will actually be arranged and stored



Physical Design

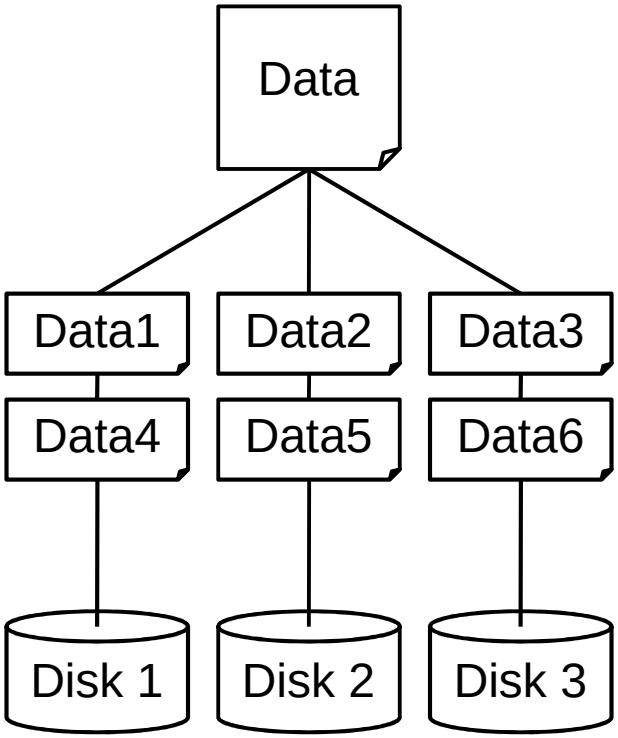
- Hardware
  - Correct storage hardware should be selected to avoid loss of data
  - Speed can also be affected by careful consideration of hardware
  - e.g. RAID Arrays?
- File Structure
  - Specifying the structure of files on disks often has a huge impact on performance
  - Implementation of these structures is also specific to a DBMS
  - Indexes can be chosen to further improve speed

RAID Arrays

- **RAID - redundant array of independent (inexpensive) disks**
  - Storing information across more than one physical disk
  - Speed - can access more than one disk
  - Robustness – disk failure doesn't always mean data is lost
- RAID Arrays are controlled by software or hardware
  - At the OS level the RAID array will appear to be a single storage device
  - The array may actually contain dozens of disks
- Different levels (RAID 0, RAID 1,…)

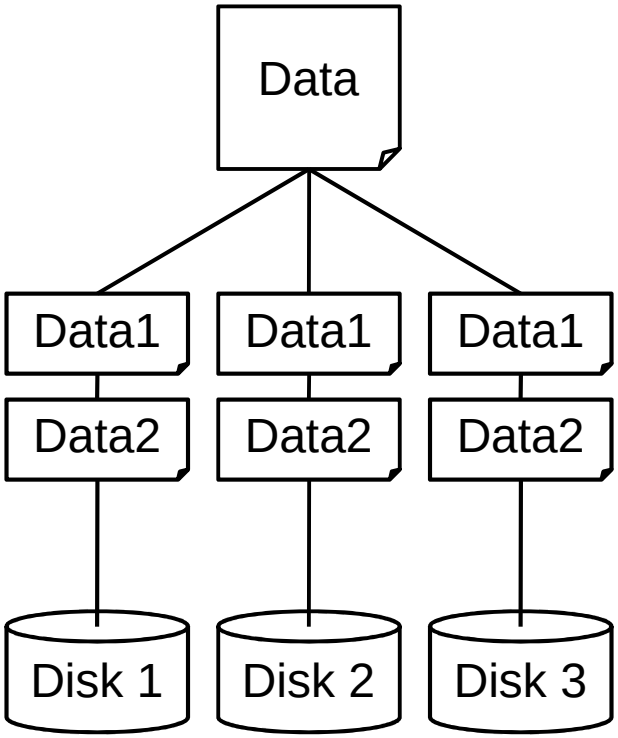
RAID Level 0

- Files are split across several disks (Striping)
  - Each file is split into parts, one part stored on each disk in the same position
  - Sizes of each part are determined by the array controller
  - Vastly improves speed, but **no redundancy**
  - If **any** disk fails, **all** data is unrecoverable
  - (No parity or mirroring)



RAID Level 1

- Files are duplicated over all disks (Mirroring)
  - **Each** file is copied onto **every** disk
  - Provides **improved read performance** but no write performance
  - **Large amounts of redundancy**, if all but a single disk fail the data can still be recovered



Parity Checking

- Above RAID 1 involves calculating parity bits
- Parity reduces the number of disks you need for redundancy
- Parity is often calculated using XOR ⊕

- For two bits, A and B, XOR is true if either A is true or B is true, but not both

A	B	⊕
0	0	0
0	1	1
1	0	1
1	1	0

Note: From ANY two columns you can work out the third!

Parity Checking

- The parity of n blocks of data is calculated as $D_1 \oplus D_2 \oplus \ldots \oplus D_n$
- For example:

D_1	0 0 1 1 0 1 1 0
D_2	1 0 1 1 0 0 1 0
D_3	1 1 0 0 0 0 0 0
D_P	0 1 0 0 0 1 0 0

XOR is 0 if there is an even number of ‘1’ bits and 1 if there is an odd number

Recovery With Parity

- If any single disk breaks, including the parity disk, XOR can be used to re-calculate that value. For example:

D_1	0 0 1 1 0 1 1 0
~~D_2~~	~~1 0 1 1 0 0 1 0~~
D_3	1 1 0 0 0 0 0 0
D_P	0 1 0 0 0 1 0 0

Recovery With Parity

- If any single disk breaks, including the parity disk, XOR can be used to re-calculate that value. For example:

D_1	0 0 1 1 0 1 1 0
D_3	1 1 0 0 0 0 0 0
D_P	0 1 0 0 0 1 0 0
D_2	1 0 1 1 0 0 1 0

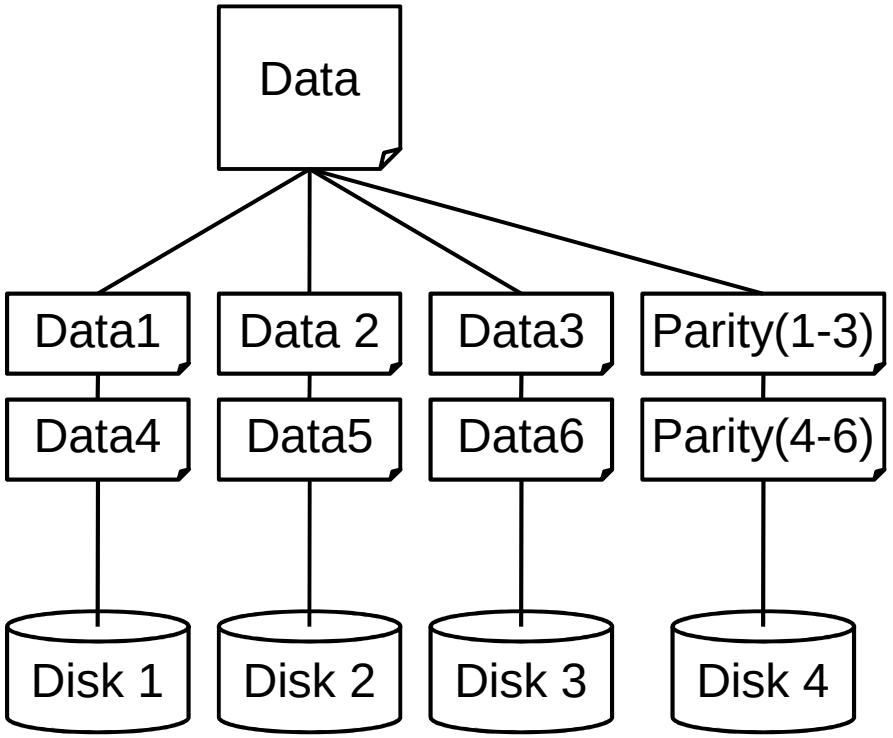
Recovery With Parity

- If any single disk breaks, including the parity disk, XOR can be used to re-calculate that value. For example:

D_1	0 0 1 1 0 1 1 0
D_2	1 0 1 1 0 0 1 0
~~D_3~~	~~1 1 0 0 0 0 0 0~~
D_P	0 1 0 0 0 1 0 0

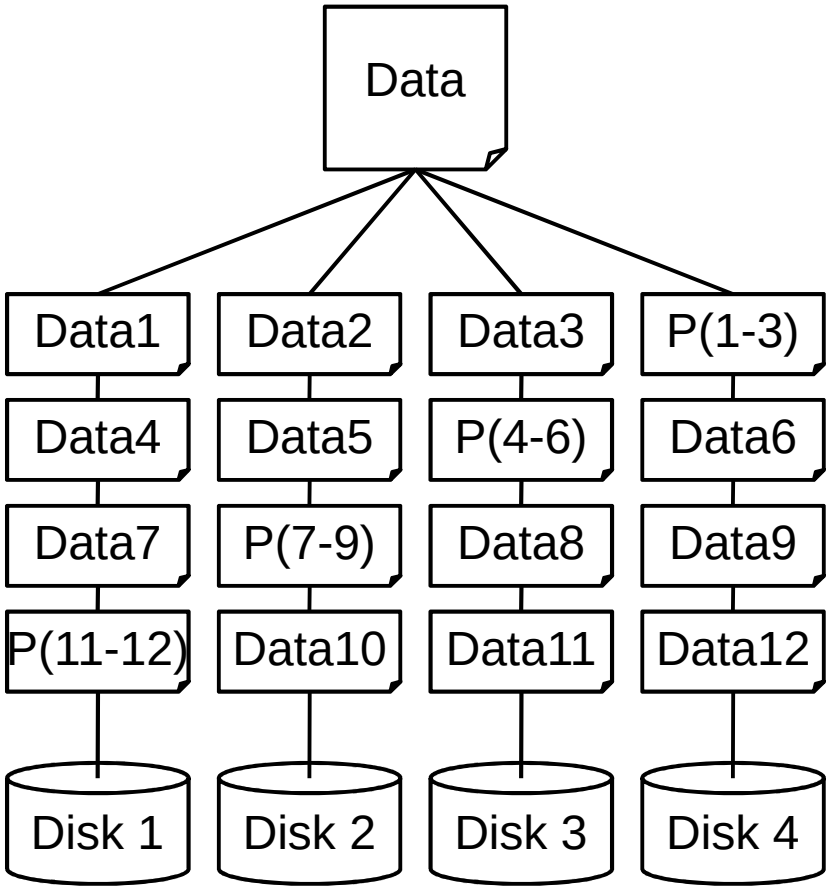
RAID Level 3

- Data is striped over all but one disk, with a parity disk for redundancy
  - Data is split into **bytes**, bytes are written to separate disks
  - The final disk stores parity information
  - Extremely fast, and will allow for 1 disk failure
  - (RAID 2: bit-level striping, RAID 4 is block-level)



RAID Level 5

- Data is striped over disks with a distributed parity
  - Data is split into **blocks**, parity blocks are distributed throughout all disks
  - Extended version of RAID 3, and will even allow continued use after 1 disk failure
- (RAID4 is RAID 5 with a single parity disk)



Other RAID Issues

- Other RAID levels:
  - Allowing more than a single disk failure with the minimum of redundancy
  - Nested RAID levels. For example RAID 10 is a mirrored array of striped arrays
- Considerations with RAID systems
  - Cost of disks
  - Do you need speed or redundancy?
  - How reliable are the individual disks?
  - ‘Hot swapping’

File Structure

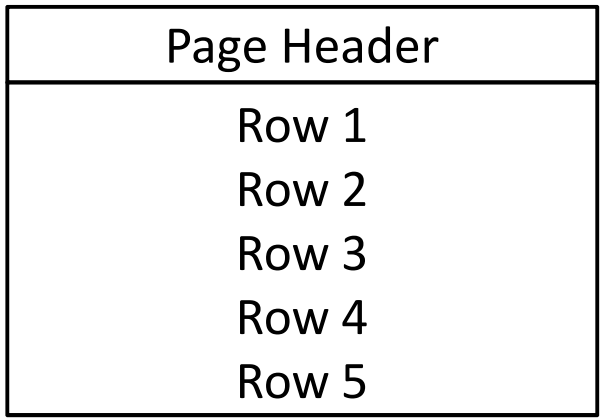
- The structure of files on a disk is separate from the RAID configuration
- **File structure is managed by the DBMS**
- Often the designer of a database can have control over aspects of this structure

- In general, file structure is concerned with:
  - How files are stored on a disk
  - In what order files are stored
  - The speed at which a file can be retrieved
  - The speed at which files can be inserted or deleted

Pages and Rows

- Row data is generally not written to disk individually. Instead, rows are grouped into pages
- Pages are often used as the atomic unit of I/O, any read or write to the disks will be a page, even if only a single row is affected

- A page will include a header and the row data:



- There are usually many rows in a page, but not always

Pages and Rows

- A table will often span multiple pages
- INSERTs may be added at the last position in the newest page
- Additional pages can be added if the previous one is full

- For a student table:

Page 1

11011465	Jack	...	1
11011658	Robert	...	2
11044348	Sarah	...	3
11051499	Max	...	1

Page 2

11012234	James	...	2
11034868	Mike	...	2
11048345	Anne	...	1

Unordered Files

- Unordered files are often called Heaps
- New records are inserted into the last page of the file
- If the page is full, a new page is added at the end of the file
- This structure makes insertion very efficient
- There is no ordering on values: searching must be conducted linearly
- To delete a record:
  - Retrieve page
  - Mark row as deleted
  - Write page back
- This space is difficult to reclaim, so performance will often deteriorate over time

Ordered Files

- Data sorted by one or more fields is called a **sequential file**
- Inserting and deleting from an ordered file is difficult
- If there is sufficient space on the correct page, a record can be inserted and that page re-written
- Full pages can propagate along and require many rewrites
- One solution is to temporarily add records to an overflow file, these are merged at a later time
- Overflow files improve inserts, but make searches more difficult

Binary Search

- A huge benefit of an ordered data structure is the concept of the binary search
- A binary search is only possible when searching for specific values in a field, where the data is also ordered by that field

- Consider the Student table, ordered by sID:

```
SELECT *
 FROM Student
 WHERE sID = 11062365;
```

- A linear search through the database could involve thousands of reads

Binary Search

Searching for: 11062365

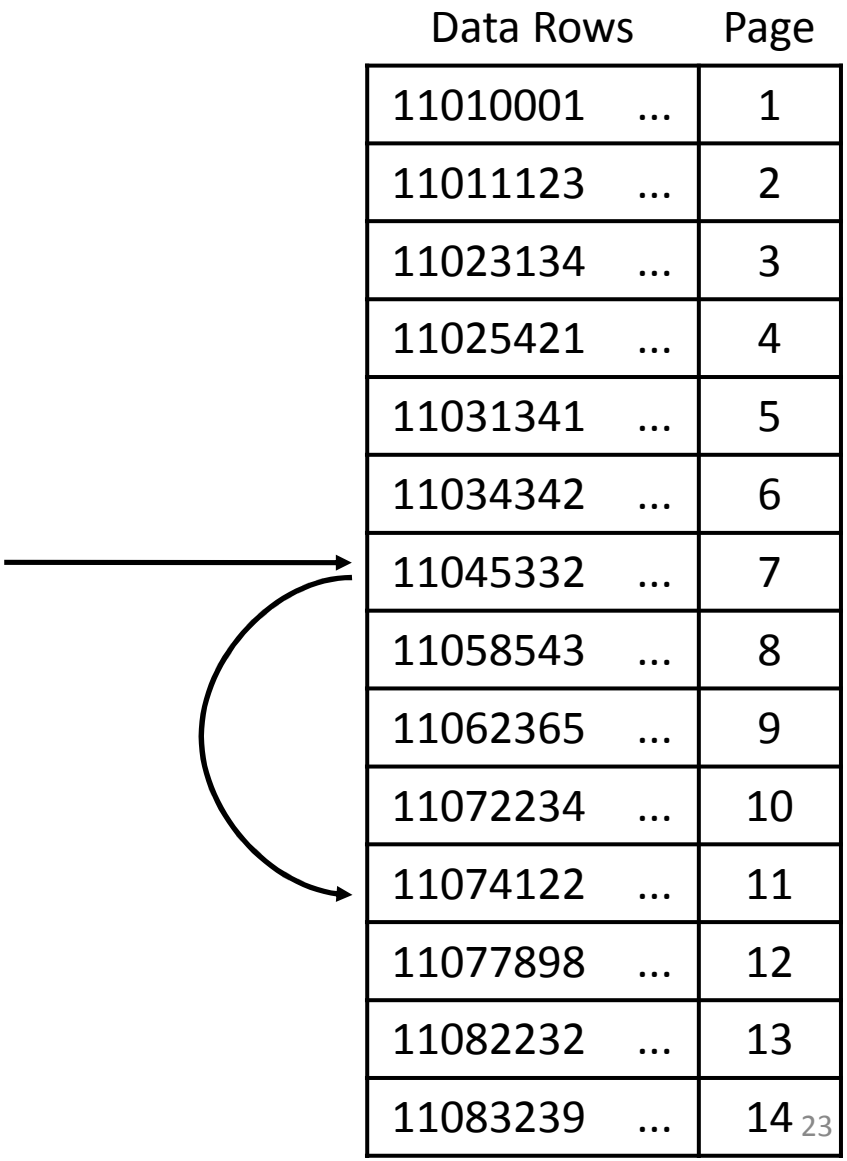
- We begin the search by retrieving the page half way through the file
- The ID values in the page are smaller than the one we're looking for so we next look further down the file

Data Rows		Page
11010001	...	1
11011123	...	2
11023134	...	3
11025421	...	4
11031341	...	5
11034342	...	6
11045332	...	7
11058543	...	8
11062365	...	9
11072234	...	10
11074122	...	11
11077898	...	12
11082232	...	13
11083239	...	14

Binary Search

Searching for: 11062365

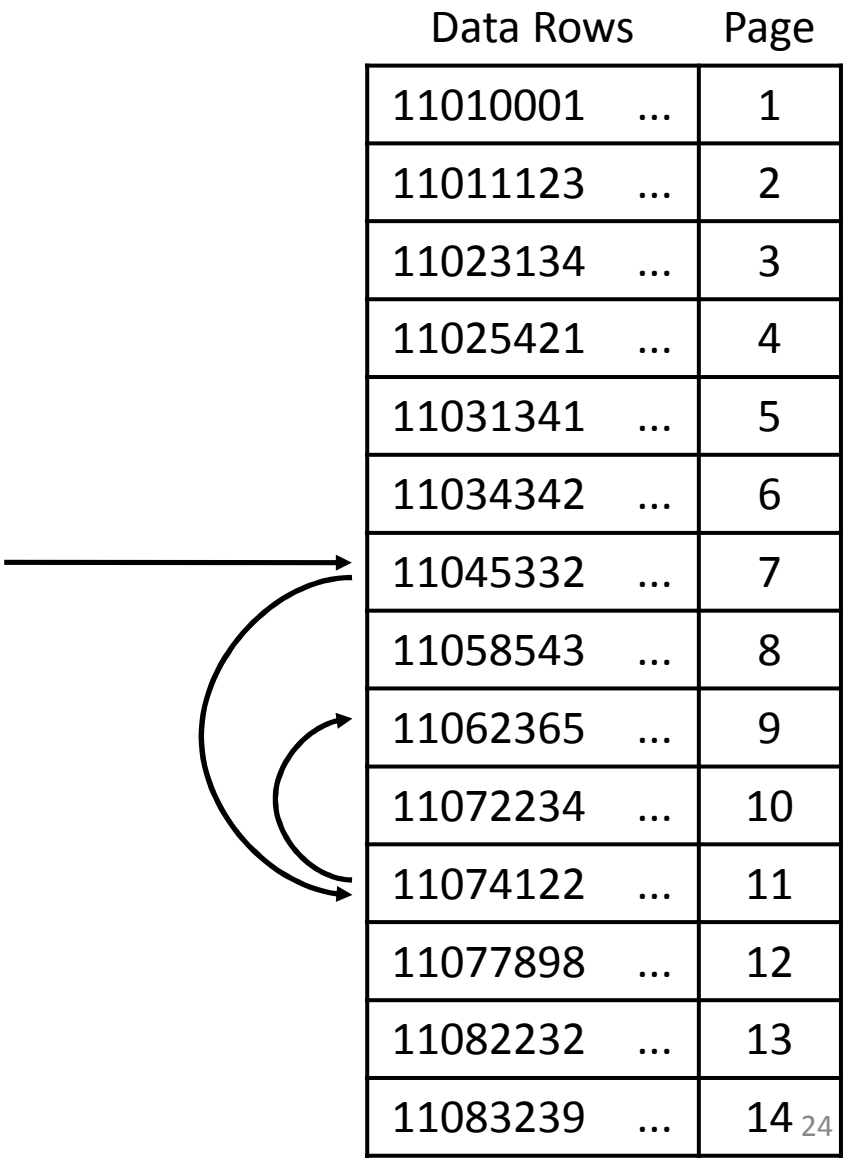
- We next retrieve the page half way through the remaining half, that contains our value
- The values are larger than the ID we are looking for, so we must travel backwards in the file



Binary Search

Searching for: 11062365

- We retrieve the page half way between the two previous pages
- The value we are looking for is on this page. We read the remaining row data
- Binary searches complete in $Log_2 n$ time, which is often better than a linear search



Indexes

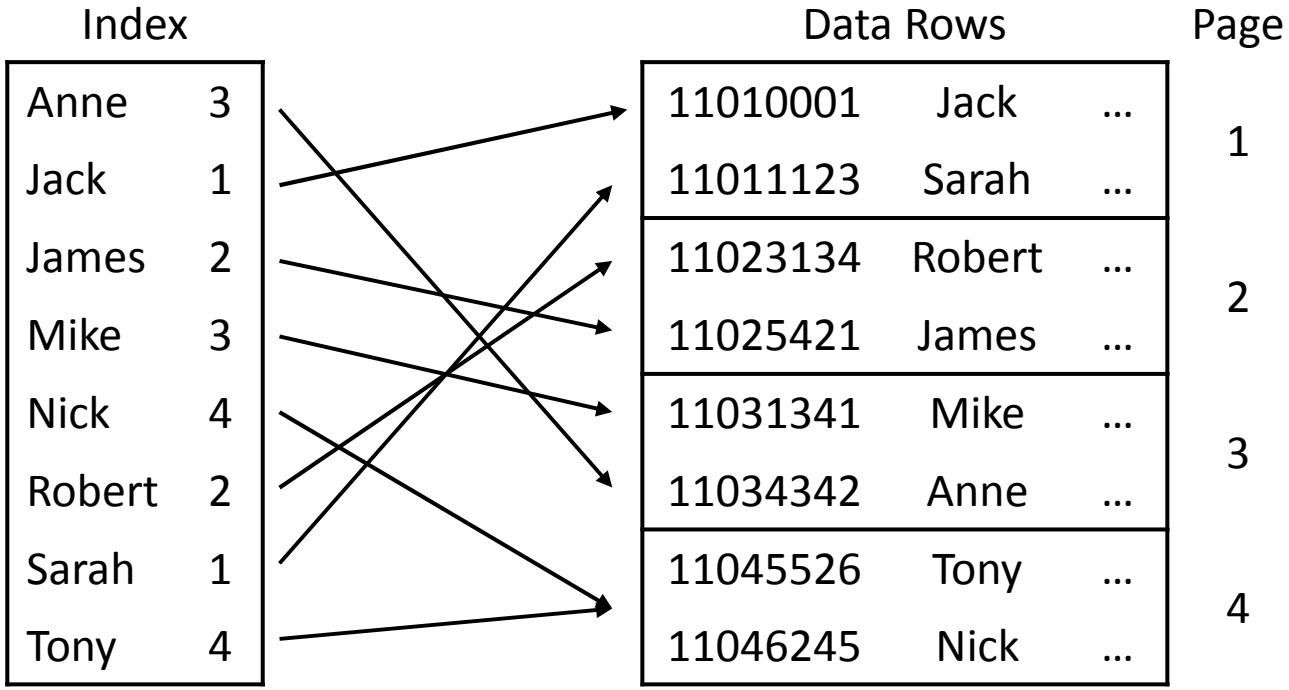
- Strictly speaking, the relational model states that the ordering of tuples does not matter
- In reality this is inefficient, searching and ordering are much easier using sequential files
- We can obtain further improvements with indexes
- An Index is a data structure that resides alongside a table, providing faster access to the rows
- An Index is associated with one of more fields, improving searches involving those fields
- The underlying data may or may not be ordered

Indexes

- Indexes are not unlike those you find in books
  - The aim is to simplify the search for key words or values
  - Often much faster than looking through the book linearly
  - The index will be ordered to improve search efficiency
- There are a number of types of indexes
  - **Primary** indexes refer to a sequential file ordered by a key (unique)
  - **Clustered** indexes refer to a sequential file ordered by some fields that *may not* be unique
  - **Secondary** indexes exists separately to the data ordering

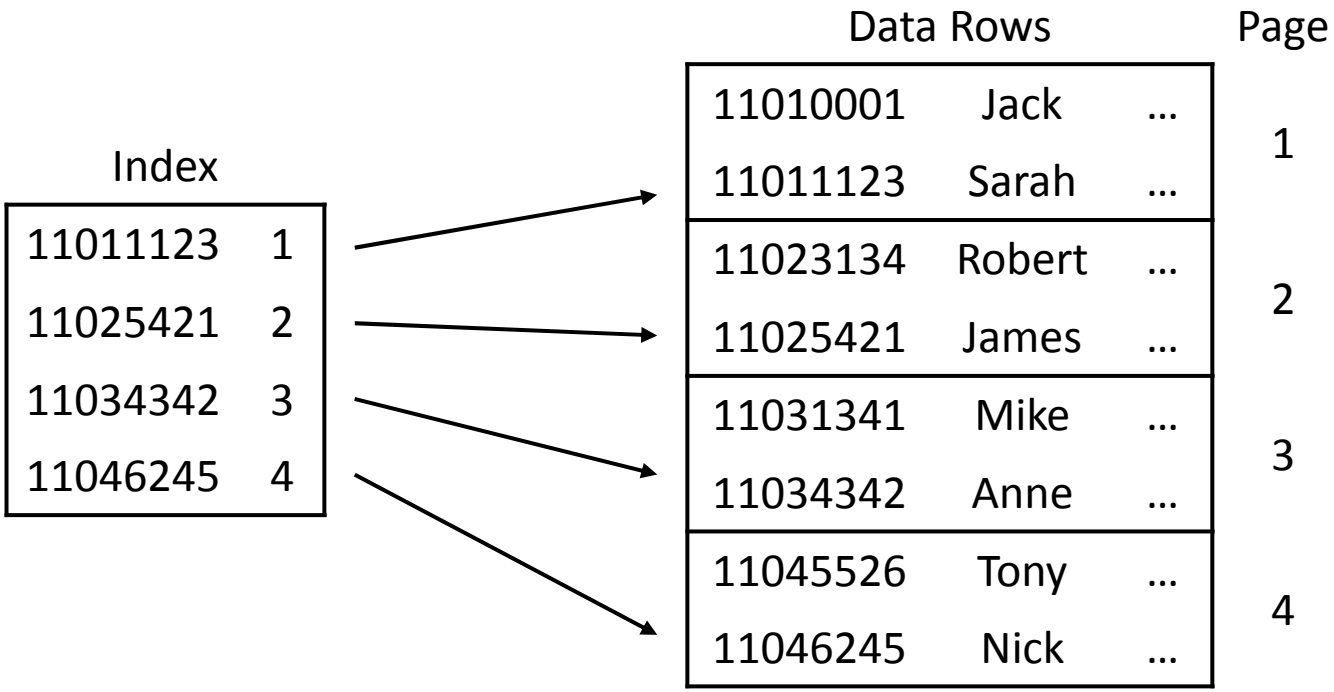
Sparse and **Dense** Indexes

- Indexes can be broadly split into two categories, sparse and dense indexes
  - **Dense indexes contain a value matching every row**
  - **Take up more memory but don't require the data to be sorted**



Sparse and Dense Indexes

- Indexes can be broadly split into two categories, sparse and dense indexes
  - **Sparse indexes only match a subset of the rows**
  - **Efficient searches using a sparse index need to be on a sequential file**

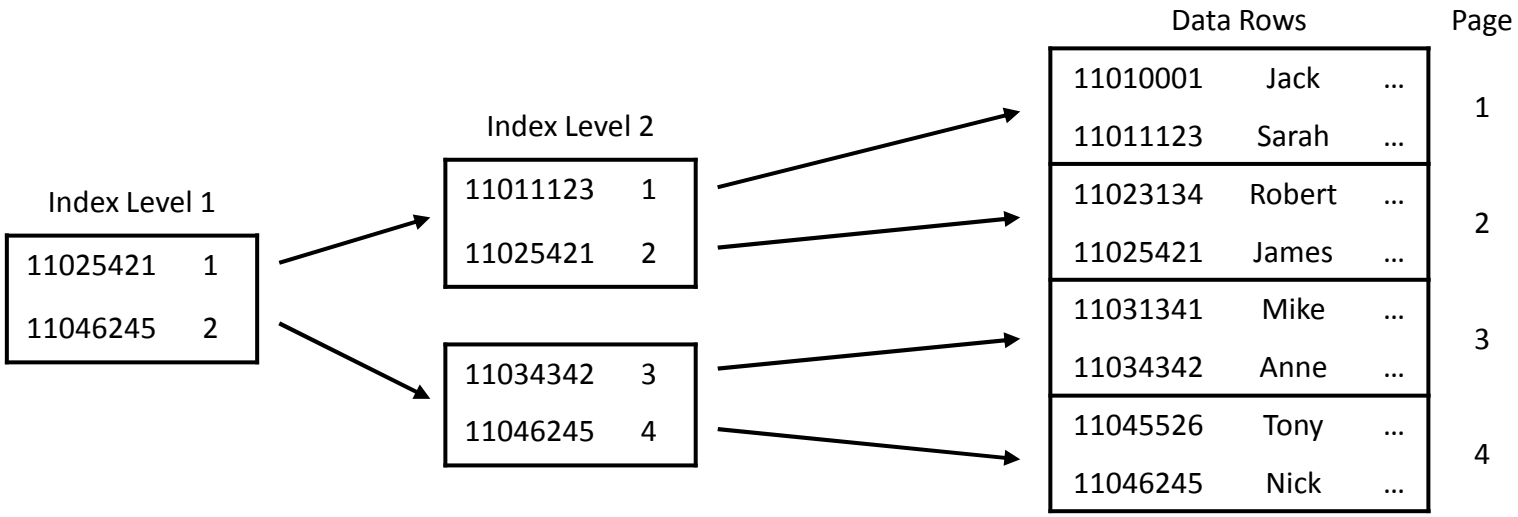


Benefits of Indexing

- Even sparse indexes significantly reduce the number of page reads required to retrieve the specific page a search requires
- Indexes are often stored in memory to further improve search speed
- However, every index must be maintained, and this adds complexity to INSERT, UPDATE and DELETE queries

Multi-level Indexes

- It's possible, and often beneficial, to add higher levels of sparse index above existing ones
- Higher levels contain fewer index rows, and point you towards ***less sparse*** indexes lower down
- The final level points you towards the table data

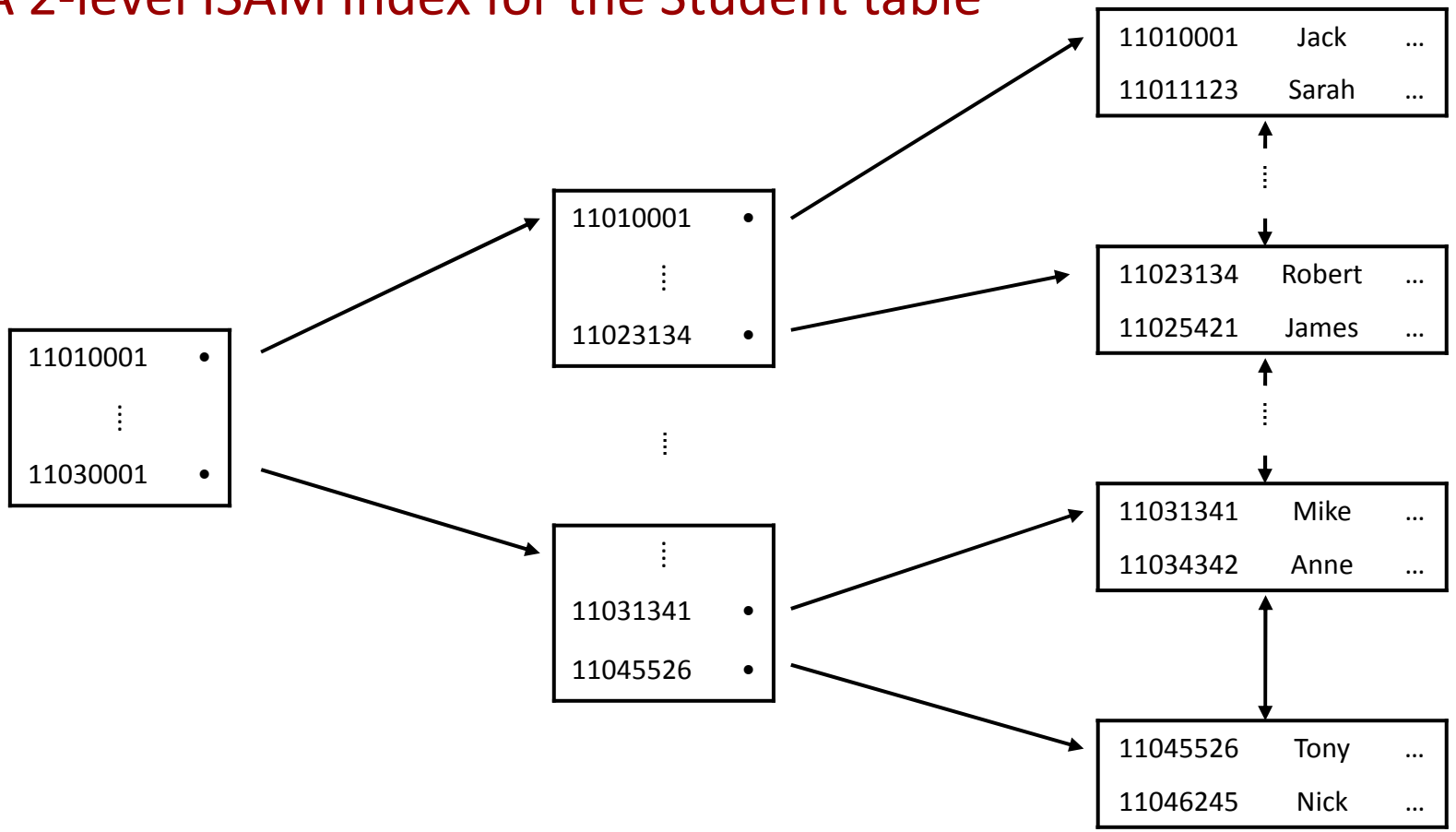


ISAM

- **ISAM stands for Indexed Sequential Access Method**
  - Essentially a static, sparse, often multi level **primary** index
  - The lowest level index points to anchor rows that represent each page. The index always references pages, rather than specific rows
  - Pages are linked together either by being consecutively stored on disk, or each page holds a pointer to the next one. This means values can be read sequentially extremely fast
  - Because the index is static, it must be reorganised occasionally to prevent a deterioration in speed
  - ISAM is less suitable for databases with very frequent inserts or deletes. Overflow areas are used if necessary

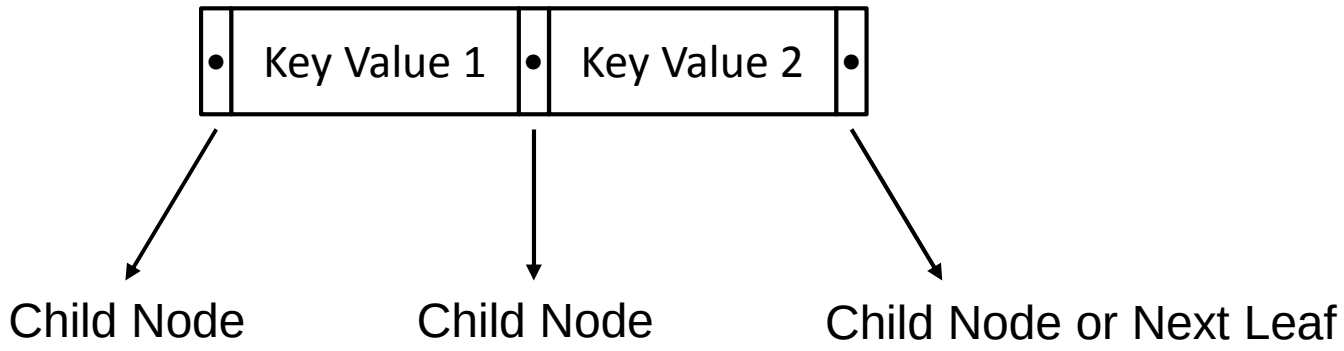
ISAM

- A 2-level ISAM Index for the Student table



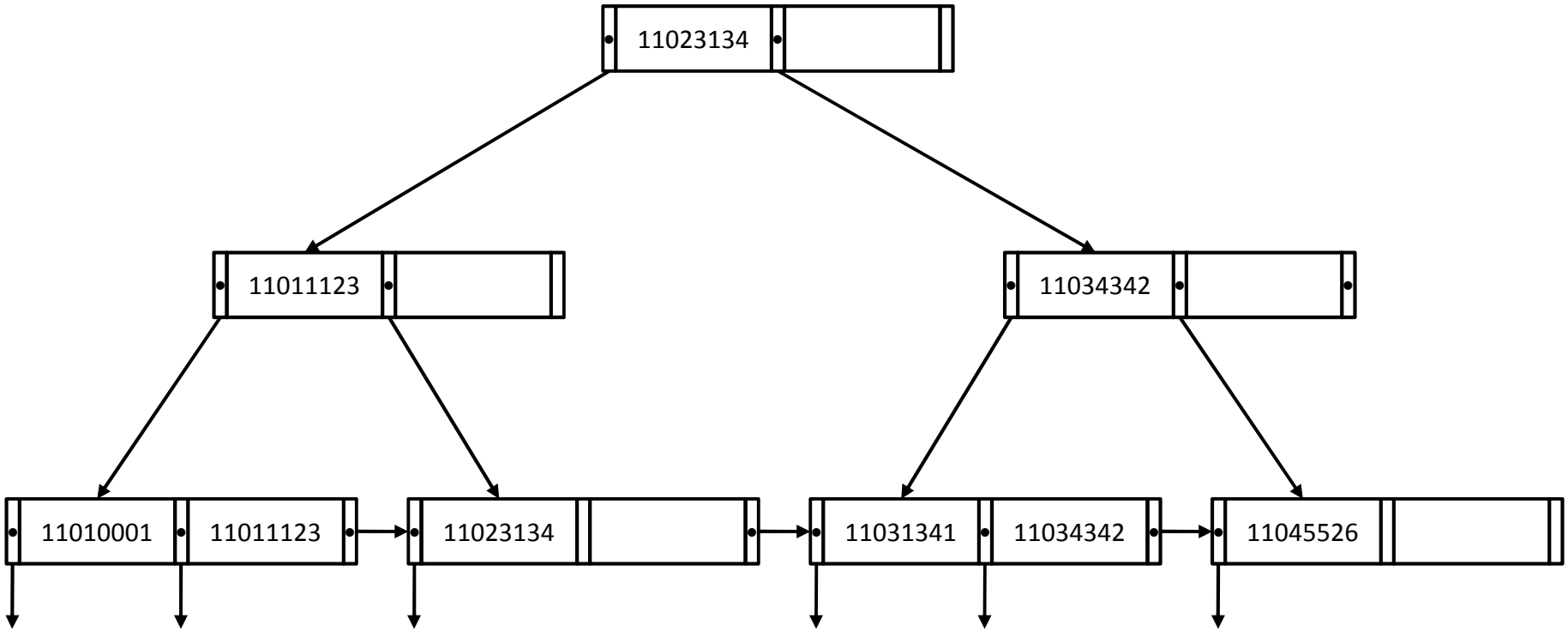
B^+-Trees

- B-Trees are balanced tree structures, and are now the most common method for indexing in databases. They are used as a dense index. (B+ trees store data only at leaf nodes)
- A balanced tree is a hierarchy of **nodes** each of which links to child nodes below it, much like ISAM
- The top of the tree is the **root**, nodes with no children are called **leaf** nodes. Leaf nodes are interconnected for sequential access
- An example node would look like this:



Example B$^+$-Tree

- An example B$^+$-Tree Index for the Student table



Benefits of B^+-Trees

- B+-Trees vastly reduce the number of reads necessary to access a specific database row
  - For example, if the order (the number of allowed children) is 256, then a single row in a database of 16million rows could be accessed with 4 disc reads
  - Inserts and Deletes usually require the tree nodes to be slightly rearranged to balance the tree. This is an efficient procedure, but we will not cover it in this module (usually covered in G52ADS)

Clustered Indexes

- In many modern DBMSs, rather than the leaf nodes of a B^+-tree pointing to the locations of the data rows, the leaf nodes themselves are adapted to hold the row data
  - This reduces a read operation where the row has to be looked up
  - This adapted B^+-tree is often called a **clustered index** and is the default storage structure in InnoDB and other DBMSs
  - By default, a table in InnoDB will be a clustered index on the primary key

Choosing Indexes

- You can only have one primary or clustered index
  - The most frequently looked-up value is often the best choice
  - Some DBMSs assume the primary key is the primary index, as it is usually used to refer to rows
- Don't create too many indexes
  - They can speed up queries, but they slow down inserts, updates and deletes
  - Whenever the data is changed, the index may need to change
- Create secondary indexes if another field might often be used for ordering or searching

Creating Indexes

- In SQL we use CREATE INDEX:

```
CREATE INDEX <index name>
   ON <table> (<columns>)
```

- Example:

```
CREATE INDEX sIndex
   ON Student(sName);
CREATE INDEX smIndex
   ON Student (sName, sMark);
```

Next Lecture

- Row-Orientated Vs Column-Orientated Databases
- Other Types of Database
  - OODBMSs and ORDBMSs
  - Distributed DBMSs
    - Example: BigTable
  - Semi-structured Data
- Further reading
  - The Manga Guide To Databases, Chapter 6
  - Database Systems, Chapters 24, 27 and 29

Modern Databases

G51DBS Database Systems

Jason Atkin

This Lecture

- Row-Orientated Vs Column-Orientated Databases
- Other Types of Database
  - OODBMSs and ORDBMSs
  - Distributed DBMSs
    - Example: BigTable
  - Semi-structured Data
- Further reading
  - The Manga Guide To Databases, Chapter 6
  - Database Systems, Chapters 24, 27 and 29

Row-vs column oriented databases

Row-orientated DBMSs

- Until now, we have focused on RDBMSs, which are row-orientated
  - Data is arranged in rows (tuples)
  - This is reflected on the disks in pages

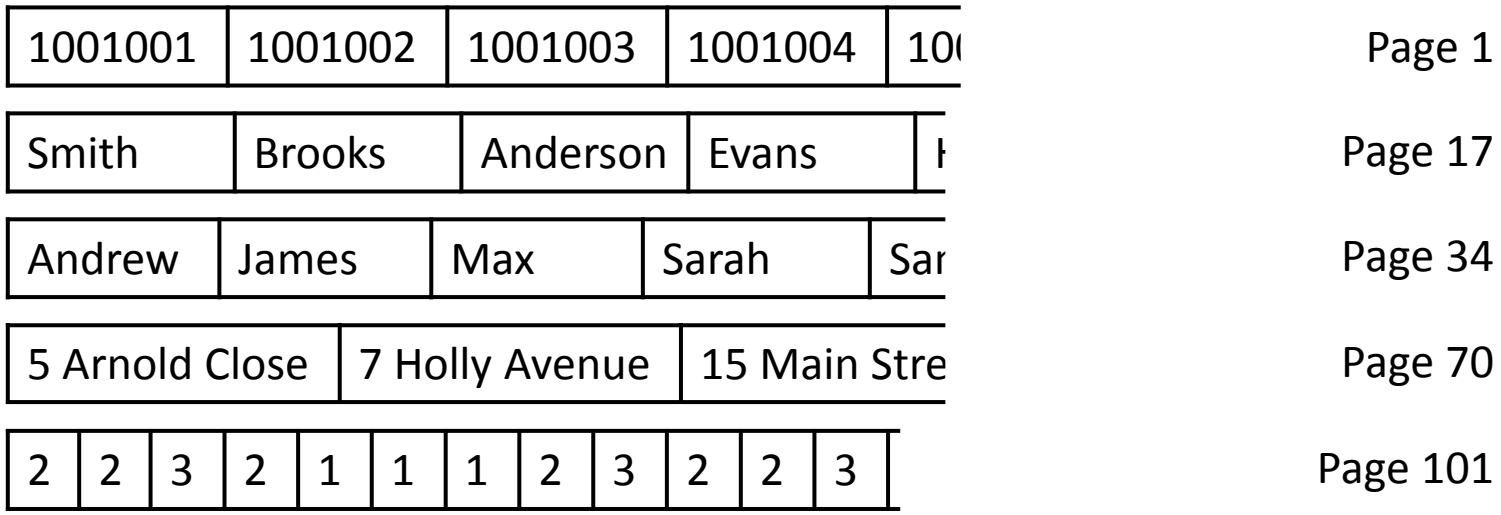
1001001	Smith	Andrew	5 Arnold Close	2
1001002	Brooks	James	7 Holly Avenue	2
1001003	Anderson	Max	15 Main Street	3

Page 1

1001004	Evans	Sarah	Flat 1a, High Street	2

Column-orientated DBMSs

- Large scale databases often store data in a column-orientated way
  - Data is arranged by column
  - This is also reflected on the disks in pages



Rows vs Columns

- Seek time is often the key factor in database speed
- It is often as quick to read multiple bytes as one byte
- You want the data you read at the same time to be in the same place (or close to it)
- The optimum storage strategy depends upon the demands on the database
- Row-orientated is extremely good when queries often require many columns in the same row
- Row-orientated provides fast inserts of complete rows
- Column-orientated is more suitable for fast analysis over single columns, e.g. Aggregate functions
- Column-orientated provides fast inserts when many rows are inserted at once

Distributed Databases

vs Centralised Databases

Distributed Databases

A distributed DB system consists of several sites

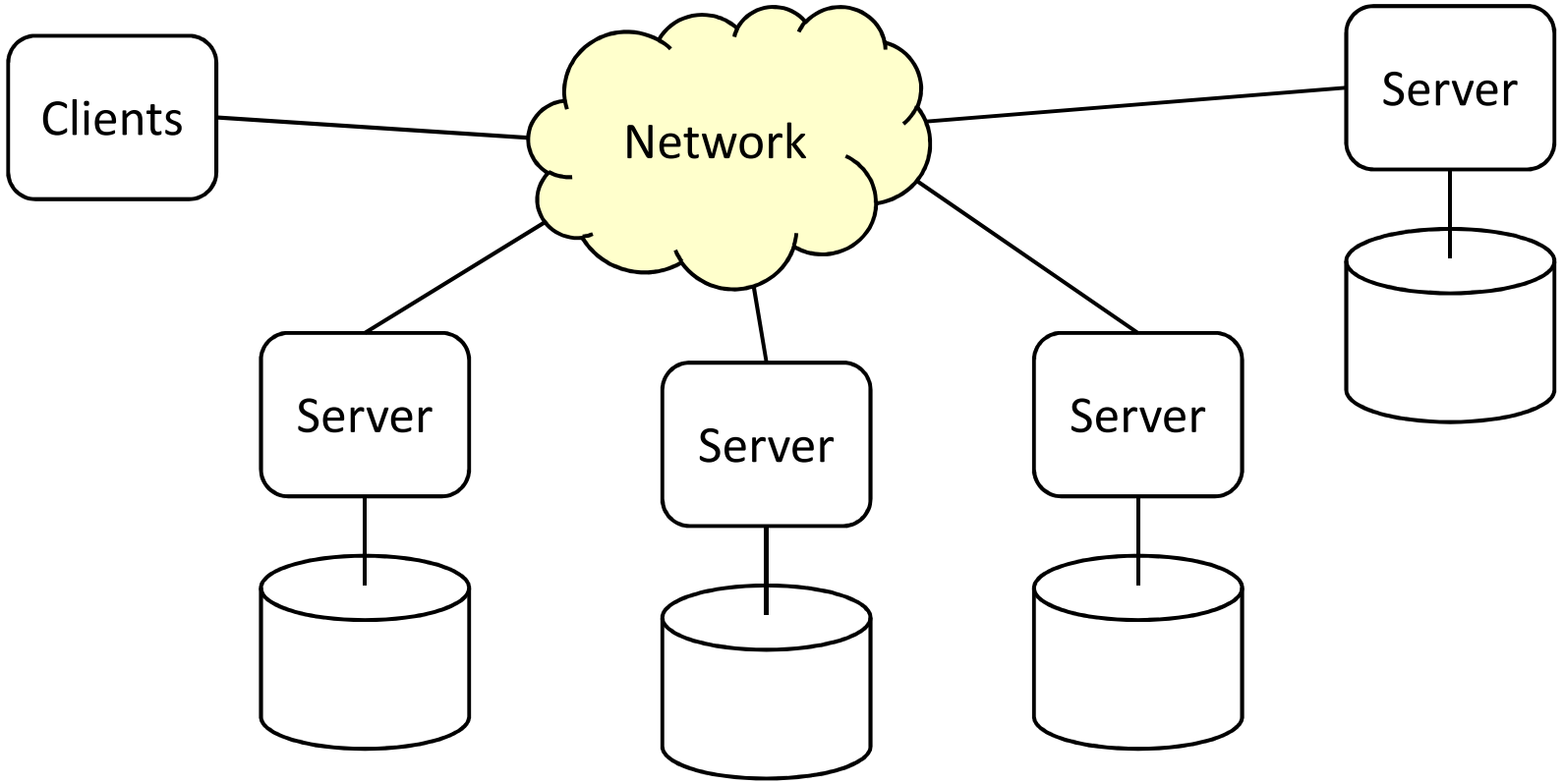
- Sites are connected by a network
- Each site can hold data and process it
- It shouldn't matter where the data is - the system is a single entity

Distributed database management system (DDBMS):

- A DBMS (or set of them) to control the databases
- Communication software to handle interaction between sites

Distributed Databases

- Distributed databases often make use of a client/server architecture



Distributed Databases

- Data split into fragments
  - Fragments may be replicated
  - Fragments/replicas allocated to sites
- Each site can:
  - Access local data direct
  - Request remote data through communications network
  - Which is used should be transparent to end user

Advantages and disadvantages

Advantages:

- Distributed processing
- Aim for locality of data
- Improves local control
  - vs a centralised database
- Integrate with local systems?
- May keep local ***copies***
- Improves reliability
  - Unlikely to all go down at once
  - Local copies an advantage?
- Easier to handle expansion?
- ***May*** be cheaper than one large database (may not)

Disadvantages:

- ***Cost?*** – multi-site maintenance, costs of communications
- Complexity – design and maintenance
- Security issues – more locations to secure
- Integrity control is more complex – duplication?
- Lack of current standards

Other types of databases

Object Oriented Database

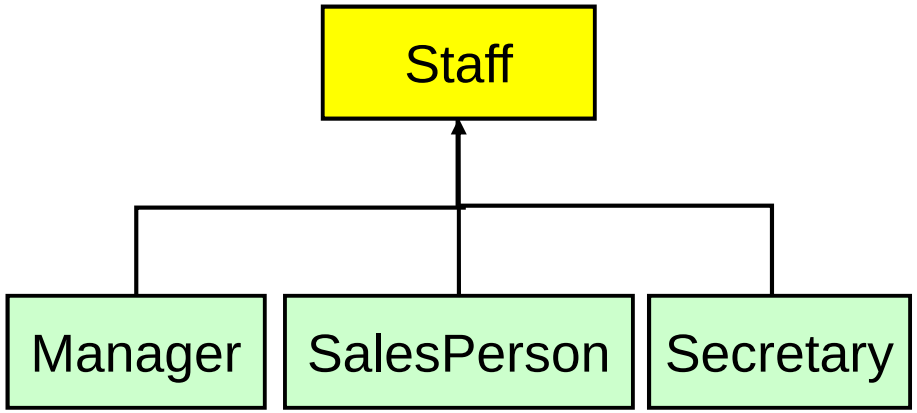
Weaknesses of RDBMSs

- Poor representation of real world objects
  - Often split up into multiple entities and relationships
  - Complex objects and relationships are not handled well
- RDBMS needs horizontal homogeneity
  - Each tuple must have the same attributes

  and vertical homogeneity
  - Values in each cell of a column must be the same type
- Often involves a great deal of normalisation…
  … and a lot of JOINs to recombine data
- Schema changes are (relatively) hard to make
- Poor support for anything beyond standard types
  - e.g. images and video: BLOB – Binary Large OBjects

Example using an RDBMS

- Base class: **Staff**: staffID, name, position, DOB, salary, etc
- Sub-classes: **Manager**: bonus, positionstartDate
  SalesPerson: salesArea, carAllowance
  Secretary: typingSpeed



Example using an RDBMS

- Base class: **Staff**: staffID, name, position, DOB, salary, etc
- Sub-classes: **Manager**: bonus, positionstartDate
  SalesPerson: salesArea, carAllowance
  Secretary: typingSpeed
- Option 1: 4 tables : Staff, Manager, SalesPerson, Secretary
  - Add staffID attribute to subclass entities
  - Need a join to recreate the objects
  - Lost the semantic information of sub-class/super-class entity
- Option 2: 3 tables : Manager, SalesPerson, Secretary
  - Repeat the base class info in each sub-class entity
  - Lost semantic information that there is a base class
- Option 3: 1 table : Staff
  - Add a type variable, but no way to limit attributes to match type
  - Many attributes which are not used unless of correct type

Example tables in an RDBMS

Staff:	**staffID**	**name**	**position**	**DOB**	**salary**
Manager:	**staffID**	**bonus**	**positionStartDate**		
Sales person:	**staffID**	**salesArea**	**carAllowance**		
Secretary:	**staffID**	**typingSpeed**			

Staff:	**staffID**	**name**	**position**	**DOB**	**salary**	**bonus**	**positionStartDate**
Sales person:	**staffID**	**name**	**position**	**DOB**	**salary**	**salesArea**	**carAllowance**
Secretary:	**staffID**	**name**	**position**	**DOB**	**salary**	**typingSpeed**	

Staff: (type field would give information about employee type)

staff ID	**name**	**type**	**position**	**DOB**	**salary**	**bonus**	**position StartDate**	**sales Area**	**carAllowance**	**typing Speed**

More weaknesses of RDBMSs

- Conversion to and from standard procedural languages often requires a lot of code
- Difficulty handling recursive relationships
  - e.g. Table: {managerId, employeeId}
  - Find all managers who directly **or indirectly** manage employee 030054
  - Direct is easy, as is manually adding each level
- No differentiation between relationships
  - No semantics are stored – not even names
  - Although could name foreign keys appropriately

Object Oriented Databases

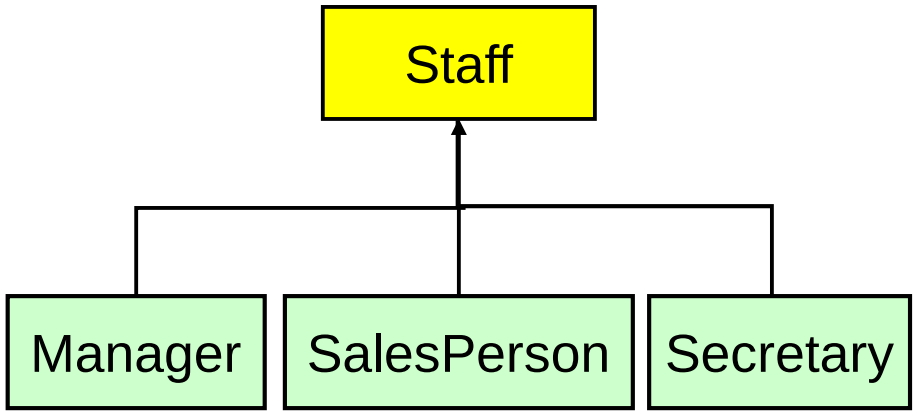
- An object oriented database (OODB) is a collection of **persistent objects**
- **Objects** - instances of a defined class
- **Persistent** - objects exist independently of any program

- An object oriented DBMS:
  - Manages a collection of objects
  - Allows objects to be made persistent
  - Permits queries to be made of the objects
  - Does all the normal DBMS things as well

As an example, try ObjectDB, free demo from http://www.objectdb.com/

OODB Example

- The earlier example:



- The database is a persistent collection of objects
  - Staff sub-class objects
- Can query the objects, by type, e.g. sum salary of all Managers, or all Staff

- OODB solution:
- Make an abstract Staff class
  - You cannot make a Staff object directly
- Make concrete Manager, SalesPerson and Secretary classes
  - You can make objects of these types
- Call persist (or equivalent) on database manager, passing it each object

Object Oriented Databases

Advantages

- Good integration with Java, C++, etc
- Has the advantages of the (familiar) object paradigm
- Better modelling of real-world objects
- Easier addition of data types and schema changes
- Can store complex information
- Fast to get whole objects
- Potentially improved performance in some cases

Disadvantages

- There is no agreed underlying data model (unlike the relational model)
- Can be more complex and less efficient
- Aimed at programmer rather than user
- Lack of support for views
- Lack of support for security
- Cannot as easily take advantage of other database front-ends

See Connolly and Begg, section 27.8

Other types of databases

Object Relational Database

Object-Relational Databases

Extend an RDBMS with object concepts:

- Data values can be objects of arbitrary complexity
  - Can include functions to operate on them
- These objects can have inheritance etc.
- You can query the objects as well as the tables

- An object relational database
- Retains most of the structure of the relational model
- Needs extensions to query languages (SQL or relational algebra)
- Most DBMSs already implement much of this as part of SQL3 onwards

Other types of databases

NoSQL databases

NoSQL

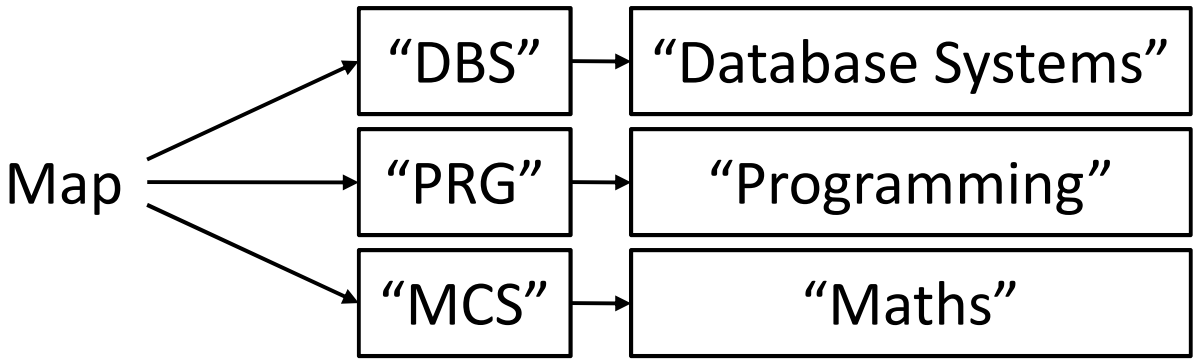
- **NoSQL: Without SQL, or 'Not Only SQL'**
- Replace (or change/enhance?) SQL as a query language
- Tend to be developments of data-oriented online companies
  - e.g. Google, Amazon, Facebook,…
- Databases usually distributed and fault tolerant
- Databases don't usually require/guarantee ACID
  - Atomicity, Consistency, Isolation, Durability
  - Often have 'eventual consistency' rather than guarantees throughout
  - Changes may take a while to 'trickle through' the system
- Often optimised for append and retrieve only
- Usually good for huge tables, needing fast access, where data can have different forms, and joins are not needed

Huge Databases

e.g. Google Bigtable

Maps

- A map is a data structure that holds <key,value> pairs (key or value can have multiple parts)
- Much like the associative arrays we saw in PHP
- Values are accessed by using the required key



Example: Google BigTable

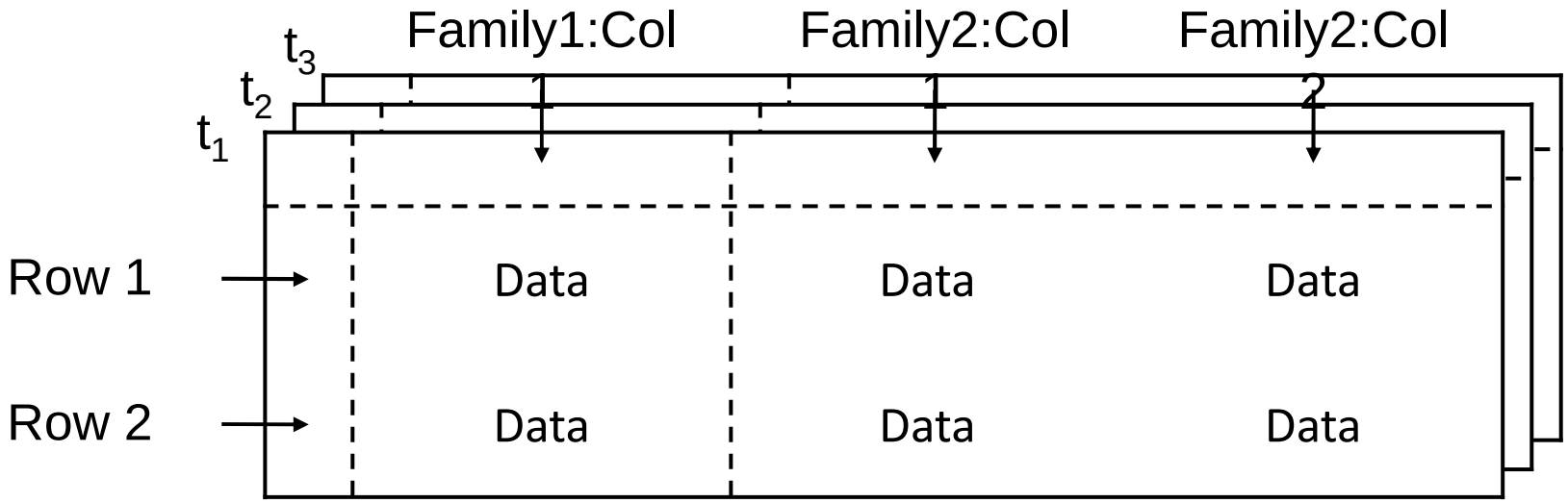
- Google BigTable is a distributed database (more or less) that contains **sparse**, **distributed**, **multi-dimensional** and **sorted** maps of data
  - Sparse: Much of the data has no value/is missing
  - Distributed: Tables are stored over many locations, and duplicated
  - Multi-dimensional: Cells are accessed by more than just a row and column
  - Sorted map : indexed by values
- The aim of BigTable is massive scalability.
- Used on Google Earth, Google Maps, Youtube etc.

  See http://research.google.com/archive/bigtable.html

  Or http://en.wikipedia.org/wiki/BigTable

BigTable Maps

- In BigTable, maps are multi-dimensional, with string keys (row, column, time) referencing a single string
- Exact use of rows and columns depends on data being stored
  - E.g. Row could be reversed URL and columns the page contents and any links to the page
- Columns are grouped into families
- Time is used to store multiple versions of data – allows removing old versions



Other types of databases

XML and semi-structured data

Semi-structured data

- Semi-structured Data :
  - A new data model designed to cope with problems of information integration
  - Where data from multiple sources could theoretically be used together, but varies wildly in structure
- XML : A standard language for describing semi-structured data schemas and representing data
- Some kind of tree structure, potentially with cross-links

XML

- XML = Extensible Markup Language
- HTML uses tags for **formatting** (e.g., “italic”)
  XML uses tags for **semantics** (e.g., “this is an address”)
- Key idea: create tag sets for a data domain and translate all data into properly tagged XML documents, obeying the rules of the schema
- **Well formed XML**: XML which is syntactically correct; tags and their nesting totally arbitrary
- **Valid XML**: XML which has DTD (document type definition); imposes some structure on the tags, but much more flexible than relational database schema
- Well-Formed XML with nested tags is exactly the same idea as trees of semi-structured data
- XML also enables non-tree structures (with references to IDs of nodes), as does the semi-structured data model

The end

of the content lectures…

Revision Lectures

- Revision lectures will take place next week
  - Exam structure to expect
  - Tactics on doing the exam (should be obvious)
  - Possible things to expect to be examined on
  - Review of the more challenging topics from the module