CO572 - Advanced Databases (60002)

Lecture 1

What is a **database management system**? A **database** is any structured collection of data points, which can be a relational table, a set, a vector, a graph, or anything along those lines. **Data management** is needed for **data-intensive applications**, we say something processes a significant amount of data if the amount of data is larger than what fits in the CPU's cache, something in the order of a few MB. We can say that below this threshold (around 5MB - 50MB), there are other factors that likely dominate performance. A **system** is made up from components that interact together to achieve a greater goal and is usually applicable to many situations.

Applications

- The scenario is at a hospital. At any given time, there are 800 patients, producing a sample per second of 5 metrics. There are also 200 doctors and nurses, who each produce a textual report every 10 minutes, and 80 lab technicians producing a structured dataset of 10 metrics every 5 minutes. Everything must be stored **reliably**, it cannot be lost after it is stored (or with a probability $p < 0.001$).
- You are developing a interactive dashboard for a global retail company. This company has stored 500GBs of sales, inventory, and customer records, and shall provide interactive access to calculated statistics. This should allow filtering of the dataset with predicates, and support the calculation of the sums of records, all with response times below a second.

Below are some examples of typical data-intensive application patterns;

- Online Transaction Processing **OLTP**
  - lots of small updates to a persistent database
  - focused on throughput (do as many updates as possible in a time-frame)
  - ACID is key (reliable)
- Online Analytical Processing **OLAP**
  - running a single data analysis task
  - focus on latency (do queries as quickly as possible)
  - **ad-hoc** queries - we don't know what they'll be
- Reporting
  - running many analysis tasks in a fixed time budget
  - focused on resource efficiency - if we can do the same task by the same time it's due with fewer resources, then it would be cheaper
  - queries are known in advance (and can be compiled into the system)
- Hybrid Transactional / Analytical Processing **HTAP**
  - a mix of **OLTP** and **OLAP**
  - small updates woven with larger analytics
  - common application is fraud detection

Data-intensive Application vs Management System

A **application** is not generic - it's often domain-specific with the logic baked in, and is therefore hard to generalise to other applications. Generally, the cost (such as adapting for other applications) of application-specific data management outweighs the benefits. We can generalise the following (from an application to a management system);

- *Yelp*
- mobile app for geo-services
- library to manage unordered collections of tagged coordinates
- spatial data management library
- relational database
- block storage system

Requirements of a Data Management System

A data management system should fulfil the following requirements;

- **efficiency** should not be significantly slower than hand written applications
- **resilience** should recover from problems, such as power outage, hardware or software failures
- **robustness**

  should have predictable performance; a small change in the query should not lead to major changes in performance
- **scalability**

  should make efficient use of available resources - increase in resources should lead to an improvement of performance
- **concurrency**

  should transparently serve multiple clients transparently (without impacting results)

Solutions

DBMSs often provide some ingenious solutions;

- **Physical and Logical Data Model Separation**

  We typical provide a **logical data model** to the user, as they often send data in a **fire and forget** manner. The user doesn't typically care about file format, storage devices, nor portability. The DMBSs can then separate external from the internal model and therefore exploit these degrees of freedom for performance. For example, if the user doesn't care where the data is stored, we can keep the hot data on an SSD and the cold data on disk or even tape.
- **Transparent Concurrency: Transactional Semantics**
  - **Atomic** run completely or not at all (if aborted, everything is reverted)
  - **Consistent** constraints must hold before and after (can be inconsistent between)
  - **Isolated** run like you were alone on the system
  - **Durable** after a transaction is committed **nothing** can undo it
- **Ease of Use: Declarative Data Analysis**

  Retrieving information about data should be done by describing the result and the system will generate it. This can be a single tuple, some statistics, a detailed generated report, or even training a model to predict data.

However, they are not to be used as a filesystem. Similarly, it cannot be used as runtime for applications - there are support for user-defined functions, but should not be used a such. They also should not be used to store intermediate data (such as whether a user is logged in).

Relational Algebra

A schema is the definition of the attributes of the tuples in the relations, but also can contain integrity constraints.

- **vector** ordered collection of objects of the same type
- **tuple** ordered collection of objects of different type
- **bag** unordered collection of objects of the same type
- **set** unordered collection of unique objects of the same type

Relational algebra is used to define the semantics of operations and is used for logical optimisation. However, it's not actually that useful for end-users. A relation is an array which represents an n-ary relation R, with the following properties;

- each row represents an n-tuple of R
- the order of rows doesn't matter
- all rows are distinct (combined with above is a set)
- the order of columns matters - all rows have the same schema
- each column has a label (defines the schema of our relation)

Relations are **almost** sets of tuples. A rough implementation in C++ is as follows;

```
template <typename... types>
struct Relation {
    using OutputType = tuple<types...>;
    set<tuple<types...>> data;
    array<string, sizeof...(types)> schema;
    Relation(){};
    Relation(array<string, sizeof...(types)> schema, set<tuple<types...>> data):
       schema(schema), data(data) {}
};
```

A relation can then be written as follows;

```
auto createCustomerTable() {
    Relation<int, string, string> customer(
        {"ID", "Name", "ShippingAddress"}, // labels of the attribute
        {{1, "james", "address 1"},
         {2, "steve", "another address}});
    return customer;
}
```

A **relational expression** is composed from **relational operators**, and will often be referred to as a **(logical) plan**. **Cardinality** is the number of tuples in a set. Relational operations are set-based, and therefore order-invariant and duplicates are eliminated. Additionally, it's **closed**;

- every operator produces a relation as an output
- every operator accepts one or two relations as input
- simplifies the composition of operators into expressions (however expressions can be invalid)

Relational operators can be implemented as follows;

```
template <typename... types> struct Operator : public Relation<types...> {}; // therefore an operator is a relation
```

A minimal set of relational operators is as follows;

- **project** (π)
  - extract one or more attributes from a relation
  - preserves relational semantics
  - changes schema

  For example, given a table;

  table1

field1	field2	field3
A	B	C
D	E	F
G	E	I

$\pi_{\texttt{field2}}$table1

field2
B
E

  The cardinality of the output of a projection can only be determined by evaluating it, as duplicates can be eliminated. On the other hand, the upper bound of the cardinality of the output is the cardinality of the input.

  It can also extract one or more attributes from a relation and perform a scalar operation on them.

```
template <typename InputOperator, typename... outputTypes>
struct Project : public Operator<outputTypes...> {
    InputOperator input;

    variant<function<tuple<outputTypes...>(typename InputOperator::OutputType)>,
            set<pair<string, string>>>
        projections;

    Project(InputOperator input, function<tuple<outputTypes...>(typename InputOperator::OutputType)> projections : input(input), projections(projections) {};

    Project(InputOperator input, set<pair<string, string>> projections : input(input), projections(projections) {};
};

void projectionExample {
    auto customer = createCustomerTable();

    auto p1 = Project<decltype(customer), string>(customer, [](auto input) { return get<1>(input); });

    auto p2 = Project<decltype(customer), string>(customer, {{"Name", "customerName"}});
}
```

- **select** (σ)
  - produces a new relation containing tuples which satisfy a condition
  - does not change schema

– changes cardinality (number of tuples in a relation)

For example, given a table;

table1

field1	field2	field3
A	B	C
D	E	F
G	E	I

$\sigma_{\texttt{field2=E}}$table1

field1	field2	field3
D	E	F
G	E	I

The cardinality can only be determined by evaluating it, and the upper bound of the cardinality is also the cardinality of the input.

```
enum class Comparator { less, lessEqual, equal, greaterEqual, greater };

struct Column {
    string name;
    Column(string name) : name(name) {};
};
using Value = variant<string, int float>;

struct Condition {
    Column leftHandSide;
    Comparator compare;
    variant<Column, Value> rightHandSide;

    Condition(Column leftHandSide, Comparator compare, variant<Column, Value>
        rightHandSide): leftHandSide(leftHandSide), compare(compare),
       rightHandSide(rightHandSide) {};
};
```

- **cross product** ($\times$)
  - takes two inputs
  - produces a new relation by combining every tuple from the left with every tuple from the right
  - changes the schema

For example, given tables;

table1

field1	field2
A	B
G	E

table2

fieldA	fieldB
2	6
5	1

table1 $\times$ table2

field1	field2	fieldA	fieldB
A	B	2	6
A	B	5	1
G	E	2	6
G	E	5	1

The cardinality of the output is the product of the two input cardinalities.

```
template <typename LeftInputOperator, typename RightInputOperator>
struct CrossProduct : public Operator<Concat<typename LeftInputOperator::
   OutputType, typename RightOutputOperator::OutputType>> {
    LeftInputOperator leftInput;
    RightInputOperator rightInput;
    CrossProduct(LeftInputOperator leftInput, RightInputOperator  rightInput)
       : leftInput(leftInput), rightInput(rightInput) {};
};
```

- **union** ($\cup$)
  - produces a new relation from two relations containing any tuple that is present in either
  - does not change the schema (but does require schema compatibility)
  - changes cardinality

  No example provided, because it's quite simple.

```
template <typename LeftInputOperator, typename RightInputOperator>
struct Union : public Operator<typename LeftInputOperator:::OutputType> {
    LeftInputOperator leftInput;
    RightInputOperator rightInput;

    Union(LeftInputOperator leftInput, RightInputOperator rightInput):
       leftInput(leftInput), rightInput(rightInput){};
};
```

  The cardinality can only be known by evaluating (due to duplicates), and the upper bound is the sum of the cardinalities of the input.

- **difference** ($-$)
  - produces new relation from two relations containing tuples present in the first but not the second
  - doesn't change schema (requires compatibility)
  - changes cardinality

  No example provided, because it's quite simple.

```
template <typename LeftInputOperator, typename RightInputOperator>
struct Difference : public Operator<typename LeftInputOperator:::OutputType>
   {
    LeftInputOperator leftInput;
    RightInputOperator rightInput;

    Difference(LeftInputOperator leftInput, RightInputOperator rightInput):
       leftInput(leftInput), rightInput(rightInput){};
};
```

- **group aggregation** (Γ)
  - produces new relation from one input by grouping tuples that have equal values in some attributes and aggregate others - groups are defined by the set of grouping attributes (can be empty), and the aggregates are defined by the set of **aggregations** which are triples consisting of;
    - **input** attribute
    - **aggregation function** (min, max, avg, sum, count)
    - **output** attribute
  - this changes both the schema and cardinality

  For example, given the following table;

customer

ID	Name	City
1	james	London
2	steve	London
3	kate	Manchester

$\Gamma((\texttt{City}), ((\texttt{ID}, \texttt{count}, \texttt{c})))\texttt{Customer}$

City	c
London	2
Manchester	1

```
enum class AggregationFunction { min, max, sum, avg, count };

template <typename InputOperator, typename... Output>
struct GroupedAggregation : public Operator<Output...> {
    InputOperator input;
    set<string> groupAttributes;
    set<tuple<string, AggregationFunction, string>> aggregations;
    GroupedAggregation(InputOperator input, set<string> groupAttributes, set<
        tuple<string, AggregationFunction, string>> aggregations): input(input
        ), groupAttributes(groupAttributes), aggregations(aggregations){};
};
```

- **Top-N** (T)
  - produce new relation from one input selecting the tuples with the N greatest values with respect to an attribute
  - changes cardinality, but maintains schema

  For example, given the following table;

customer

ID	Name	City
1	james	London
2	steve	London
3	kate	Manchester

$T_{(2,\texttt{ID})}$Customer

ID	Name	City
2	steve	London
3	kate	Manchester

```
template <typename InputOperator>
struct TopN : public Operator<typename InputOperator::OutputType> {
    InputOperator input;
    size_t N;
    string predicate;
    TopN(InputOperator input, size_t N, string predicate): input(input), N(N)
        , predicate(predicate){};
};
```

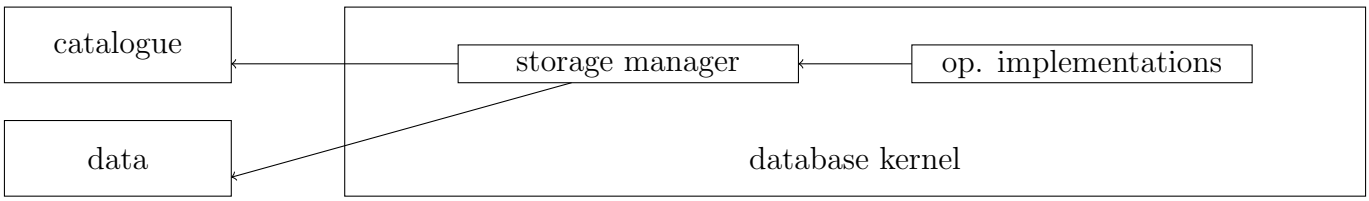
Since relational algebra is closed, operators can be combined as long as signatures are respected (cross product takes two inputs, whereas selections and projections take one);

$$\pi_{\texttt{BookID}}(\sigma_{\texttt{Order.ID == OrderedItem.OrderID}}(\texttt{Order} \times \texttt{OrderedItem}))$$

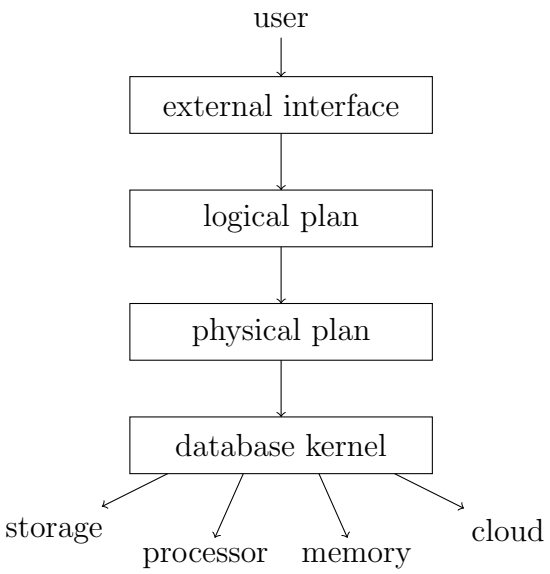
Lecture 2

Architecture

The logical plan is the relational algebra discussed last lecture and the database kernel actually interacts with the resources mentioned. A database kernel is a library of core functionality, including I/O, memory management, operators, etc. Generally, it provides an interface to subsystems (similar to an OS kernel). A basic database kernel architecture is as follows (assuming both catalogue and data live in memory).



The DBMS architecture is as follows;



The kernel interface, in C++, would look like the following;

```
class StorageManager;
class OperatorImplementations;

// just provides an interface to the above
class DatabaseKernel {
    StorageManager& getStorageManager();
    OperatorImplementations& getOperatorImplementations();
}
```

Storage

The first operation every database needs to support inserting new tuples. These inserts are usually not optimised (generally) - they arrive at the storage layer as intact tuples. Consider the following example, in SQL;

```
CREATE TABLE Employee (
    id int,
    name varchar,
    salary int,
    joiningDate int
);

INSERT INTO Employee VALUES(1, 'james', 100000, 43429342);
SELECT * FROM Employee WHERE id=1;
DELETE FROM Employee WHERE name='james';
```

The equivalent in C++ would be as follows;

```
StorageManager().getTable("Employee").insert({1, "james", 100000, 43429342});
StorageManager().getTable("Employee").findTuplesWithAttributeValue(id, 4);
StorageManager().getTable("Employee").deleteTuplesWithAttributeValue(name, "james
    ");
```

The storage manager interface would be as follows;

```
struct Table;
class StorageManager {
```

```
    map<string, Table> catalogue;

    public:
        Table& getTable(string name) {return catalogue[name]; };
};
```

In order to store tuples, we need to decide where and how we store data. Generally, for where we store it, we mainly consider disk or memory. The way we store data can vary from being sorted, compressed, RLE-encoded, etc. A simple table interface could be as follows;

```
struct Table {
    using AttributeValue = variant<int, float, string>;
    using Tuple = vector<AttributeValue>;

    void insert(Tuple) {};
    vector<Tuple> findTuplesWithAttributeValue(int attributePosition,
       AttributeValue value) {};
    void deleteTuplesWithAttributeValue(int attributePosition, AttributeValue
       value) {};
};
```

There is a fundamental mismatch in storing tuples; relations are two-dimensional, whereas memory is one-dimensional, therefore tuples need to be linearized in one of the two mainstream strategies;

- **The N-ary Storage Model** (NSM)

  In this model, the entries added are stored one after the other, in a one-to-one mapping onto memory. This is also referred to as a **row store**. An example of this is as follows;

  ```
  struct NSMTable : public Table {

      // not the tuple exposed to the outside
      struct InternalTuple {
          Tuple actualTuple;
          bool deleted = false;
          InternalTuple(Tuple t) : actualTuple(t) {};
      }
      vector<InternalTuple> data;

      void insert(Tuple);
      vector<Tuple> findTuplesWithAttributeValue(int attributePosition,
         AttributeValue value) {};
      void deleteTuplesWithAttributeValue(int attributePosition, AttributeValue
          value) {};
  };

  // declared as member of a class (just append to a vector)
  void NSMTable::insert(Tuple t) { data.push_back(t); }

  vector<Table::Tuple> NSMTable::findTuplesWithAttributeValue(int
     attributePosition, AttributeValue value) {
      vector<Tuple> result;
      for (size_t i = 0; i < data.size(); i++) {
          if (data[i].actualTuple[attributePosition] == value && !data[i].
             deleted) {
              result.push_back(data[i].actualTuple);
          }
  ```

```
    }
    return result;
}

// we don't want to move everything therefore we just mark them as deleted
void NSMTable::deleteTuplesWithAttributeValue(int attributePosition,
    AttributeValue value) {
    for (size_t i = 0; i < data.size(); i++) {
        if (data[i].actualTuple[attributePosition] == value) {
            data[i].deleted = true;
        }
    }
}
```

However, this may not always be a good idea, as operations will require us going over the entire vector. Recall that we are assuming everything is in memory, however locality still matters (since the data may be further apart depending on the size of the tuple).

Memory is organised in **cache lines** (usually 64 bytes in size). When a core needs something, it asks L1, L2, and L3 cache first, before going to memory, and retrieves the entire cache line that the data resides on (and cache into L1 cache). If we then access something on the same cache line, then the access is almost free (in terms of time). More tuples will therefore fit on a single cache line if they are smaller. Cache lines can also be referred to as **blocks** or **pages**.

N-ary storage works well in inserting a new tuple, when we are inserting a tuple onto the same cache line. It also works well when we want to retrieve a tuple by its index. On the other hand, it's suboptimal when we want to check every row, which in the worst case will be across different cache lines.

- **Decomposed Storage Model** (DSM)

  On the other hand, in this model, entries are stored with the columns one after the other. This is also referred to as **column store**. An example of this implementation is as follows;

```
struct DSMTable : public Table {
    using Column = vector<AttributeValue>;
    vector<Column> data;
    vector<bool> deleteMarkers;

    void insert(Tuple);
    vector<Tuple> findTuplesWithAttributeValue(int attributePosition,
        AttributeValue value) {};
    void deleteTuplesWithAttributeValue(int attributePosition, AttributeValue
        value) {};
};

// inserts cause tuple decomposition
void DSMTable::insert(Tuple tuple) {
    for (int i = 0; i < tuple.size(); i++) {
        data[i].push_back(tuple[i]);
    }
}

// find requires tuple reconstruction
vector<Table::Tuple> DSMTable::findTuplesWithAttributeValue(int
    attributePosition, AttributeValue value) {
```

```
    vector<Tuple> result;
    for (size_t i = 0; i < data[attributePosition].size(); i++) {
        if (data[attributePosition][i] == value && !deleteMarkers[i]) {
            Tuple reconstructedTuple;
            for (int column = 0; column < data.size(); column++) {
                reconstructedTuple.push_back(data[column][i]);
            }
            result.push_back(reconstructedTuple);
        }
    }
    return result;
}

void DSMTable::deleteTuplesWithAttributeValue(int attributePosition,
    AttributeValue value) {
    for (size_t i = 0; i < data.size(); i++) {
        if (data[attributePosition][i] == value) {
            deleteMarkers[i] = true;
        }
    }
}
```

Decomposed storage works well in the case when we are iterating over one column of a tuple. However, if these entries are one after the other, there is perfect data locality, which minimises the number of cache lines we need. On the other hand, insertion is suboptimal as we need to spread a tuple over memory. Similarly, accessing a single tuple, even when we know where it is in memory is suboptimal, as the tuple will need to be reconstructed.

- Hybrid Delta / Main Storage

```
struct HybridTable : public Table {
    DSMTable main; // every tuple will eventually end up here
    NSMTable delta; // will be inserted here then merged into main

    void insert(Tuple);
    vector<Tuple> findTuplesWithAttributeValue(int attributePosition,
       AttributeValue value) {};
    void deleteTuplesWithAttributeValue(int attributePosition, AttributeValue
        value) {};
    void merge();
};

void HybridTable::insert(Tuple t) {
    delta.insert(t);
}

vector<Table::Tuple> HybridTable::findTuplesWithAttributeValue(int
    attributePosition, AttributeValue value) {
    vector<Tuple> results = main.findTuplesWithAttributeValue(
       attributePosition, value);
    vector<Tuple> fromDelta = delta.findTuplesWithAttributeValue(
       attributePosition, value);
    results.insert(results.end(), fromDelta.begin(), fromDelta.end());
    return results;
}
```

```
21
22 void HybridTable::deleteTuplesWithAttributeValue(int attributePosition,
       AttributeValue value) {
23     main.deleteTuplesWithAttributeValue(attributePosition, value);
24     delta.deleteTuplesWithAttributeValue(attributePosition, value);
25 }
26
27 void HybridTable::merge() {
28     for (auto i = 0u; i < delta.data.size(); i++) {
29         // these two operations need to atomic (otherwise we can return the
               same tuple multiple times)
30         main.insert(delta.data[i].actualTuple);
31         delta.data[i].deleted = true;
32     }
33 }
```

In conclusion, **DSM** works well fro scan-heavy queries (accessing a lot of tuples but few columns). This is common in analytical processing, and analytics mostly operate on historical data. On the other hand **NSM** works well for lookups and inserts. This is more common in transactional processing (inserting sales item, looking up products, etc). Transactions mostly operate on recent data. Hybrid exploits the aforementioned workloads, but it needs regular migrations (`merge()`) which may need to lock the database.

Catalogue

The catalogue stores metadata. The idea is that real-life data follows patterns, which if recognised and exploited, can lead to more efficiency. However metadata will need to be stored and maintained. Examples of metadata can be type, min / max values (if data is requested outside a range we know it doesn't exist), histograms, etc. Two simple ones are **sortedness** and **denseness**. Consider the following example, which is a common pattern (as the first column is often IDs);

```
1  class Table {
2      vector<Tuple> storage;
3      bool firstColumnIsSorted = true;
4      bool firstColumnIsDense = true;
5
6      public:
7          void insert(Tuple t) {
8              if (storage.size() > 0) {
9                  firstColumnIsSorted &= (t[0] >= storage.back()[0]);
10                 firstColumnIsDense &= (t[0] == storage.back()[0] + 1);
11             }
12             storage.push_back(t);
13         }
14
15         vector<Tuple> findTuplesWithAttributeValue(int attribute, AttributeValue
               value) {
16             if (attribute == 0 && firstColumnIsDense) {
17                 return { data[value - storage.front()[0]] };
18             } else if (attribute == 0 && firstColumnIsSorted) {
19                 if (binary_search(data, attribute, value)[0] == value) {
20                     return { binary_search(data, attribute, value) };
21                 }
22             }
```

```
23              ... // same scan as before
24          }
25
26          // we can also exploit the fact that order of rows doesn't matter, and
                therefore we can reorganise
27          void analyse() {
28              // this sort can be expensive (and therefore should be run regularly
                    but not always)
29              sort(storage.begin(), storage.end(), [](auto l, auto r) { return l[0]
                    < r[0] });
30              firstColumnIsSorted = true;
31              firstColumnIsDense = true;
32              for (size_t = 1; i < storage.size(); i++) {
33                  firstColumnIsDense &= storage[i][0] == storage[i - 1][0] + 1;
34              }
35          }
36  }
```

Variable Length Data

It's important to note that strings tend to have different lengths. However, we'd like to maintain fixed tuple sizes, as it allows for random access to tuples by their position. We can either overallocate space for `varchars` (for example, some system require a size parameter for maximum length), or we can store them out of place. Overallocating leads to strings under the maximum length being padded with some sort of terminator. This is good for locality and is simple to implement, however it's very wasteful for space (especially if they are too generous with lengths).

On the other hand, out of place storage now contain an index into a dictionary, which contains the string (all null terminated). However, retrieving a value will need an access to obtain the index and another to get the actual value (which has poor locality). While this is better for space (and is what most programming languages do), it's also complicated (and causes difficult garbage-collection). This gives an optimisation (**dictionary compression**) however - if we find a string that has already been used when we insert, we can use the address of the existing value for the insert.

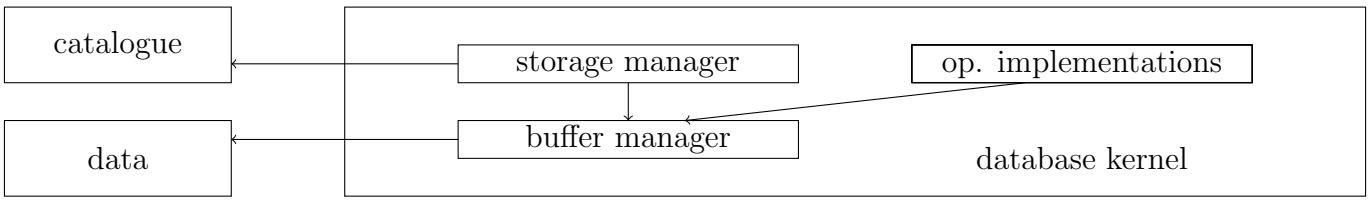
Data Storage on Disk

Disks are different from main memory in a number of ways;

- **larger pages** kilobytes compared to bytes
- **higher latency** milliseconds compared to nanoseconds
- **lower throughput** hundreds of megabytes instead of tens of gigabytes per second

  this is why we consider DBMSs as I/O bound
- **OS gets in the way** filesize can be limited, therefore DMBS need to map files and offsets

Our goals also change slightly; disks now dominate cost, and therefore complicated I/O management strategies can pay off. Due to larger pages, each page behaves like a mini-database in the case of N-ary storage. The basic kernel now looks like the following (where the catalogue is in memory, but the data is in disk);



The **buffer manager** manages disk-resident data. It maps unstructured files (large 'blobs of bytes') to structured tables for reading and writing. Since DBs don't trust OSs, it makes sure files have a fixed size, and it also safely writes data to disk when necessary. It also writes in **open (for writing) pages**. An example implementation is as follows;

```
class BufferManager;
class Table {
    BufferManager& bufferManager;
    string relationName = "Employee";

    public
        void insert(Tuple t) {
            bufferManager.getOpenPageForRelation(relationName).push_back(t);
            bufferManager.commitOpenPageForRelation(relationName);
        }

        vector<Tuple> findTuplesWithAttributeValue(int attribute, AttributeValue
           value) {
            vector<Tuple> result;
            auto pages = bufferManager.getPagesForRelation(relationName);
            for (size_t i = 0; i < pages.size(); i++) {
                auto page = pages[i];
                for (size_t i = 0; i < page.size(); i++) {
                    if (page[i][attribute] == value) {
                        result.push_back(page[i]);
                    }
                }
            }
            return result;
        }
}

struct BufferManager {
    using Tuple = vector<AttributeValue>;
    using Page = vector<Tuple>;

    size_t tupleSize;
    map<string, vector<Tuple>> openPages;
    map<string, vector<string>> pagesOnDisk; // maps one relation to many pages on
        disk (filenames)
    size_t numberOfTuplesPerPage();

    vector<Tuple>& getOpenPageForRelation(string relationName);
    void commitOpenPageForRelation(string relationName);
    vector<Page> getPagesForRelation(string, relationName);
}

vector<Tuple>& BufferManager::getOpenPageForRelation(string relationName) {
    return openPages[relationName]; // creates one if needed
}

void BufferManager::commitOpenPageForRelation(string relationName) {
    while (openPages[relationName].size() >= numberOfTuplesPerPage(tupleSize)) {
        vector<Tuple> newPage;
```

```
48
49          // move overflowing tuples to new page
50          while (openPages[relationName].size() > numberOfTuplesPerPage(tupleSize))
                {
51              newPage.push_back(openPages[relationName].back());
52              openPages[relationName].pop_back();
53          }
54
55          pagesOnDisk[relationName].push_back(writeToDisk(openPages[relationName]));
56          openPages[relationName] = newPage; // contains tuples that didn't fit on
                disk
57      }
58  }
59
60  vector<vector<Tuple>> BufferManager::getPagesForRelation(string relationName) {
61      vector<vector<Tuple>> result = { openPages[relationName] };
62      for (size_t i = 0; i < pagesOnDisk[relationName].size(); i++) {
63          result.push_back(readFromDisk(pagesOnDisk[relationName][i]));
64      }
65      return result;
66  }
```

However, we still need to determine `numberOfTuplesPerPage`;

- **unspanned pages** pages like mini-databases

  The goal of this is simplicity, and good random access performance; we want to find the record with a single page lookup, given a `tuple_id`. When we don't have enough space on a page for another tuple, we will write to disk (even when not full) and obtain a new page. However, we can quite easily infer which page a tuple will be on, since we know how many tuples are on each page. This cannot deal with large records (where the size is larger than a page), nor can we have in-page random access if the records are variable size.

  ```
  1  const long pageSizeInBytes = 4096;
  2  size_t BufferManager::numberOfTuplesPerPage() {
  3      return floor(pageSizeInBytes / tupleSizeInBytes);
  4  }
  5
  6  // space consumption of a relation of n tuples;
  7  ceil(data.size() / mnumberOfTuplesPerPage())
  ```

- **spanned pages** optimising for space efficiency

  On the other hand, the goal here is to minimise space waste and to support large records. Spanned pages can have tuples existing across pages - this is complicated and also hurts random access performance, since we can no longer easily determine where a tuple is. We can calculate the number of pages per relation as follows;

  ```
  1  long dataSizeInBytes = tupleSizeInBytes * data.size();
  2  long numberOfPagesForTable = ceil(dataSizeInBytes / pageSizeInBytes);
  ```

  However - we cannot determine the number of tuples per page as this is not constant.

- **slotted pages** random access for in-place NSM

  This is the most complicated method covered, but is also what is used by most systems. This stores tuples in in-place N-ary format, and stores the tuple count in the **page header** (at the start of the page). Offsets are also stored to every tuple, which are filled in from the **end** of the

page (hence the index of the first tuple is the last item in the page and so on). Offsets need to be typed large enough to address page;

- `byte`s for pages smaller than 256 bytes
- `short`s for pages smaller than 65,536 bytes
- `int`s for pages smaller than 4 gigabytes

Some disk-based database systems also keep a dictionary per page, which solves the problem of variable sized records (and allows for duplicate elimination, at the granularity of a single page, as previously mentioned). A global dictionary is not used as it will lead to more accesses.

Lecture 3

Join

The issue with data normalisation (see **CO130**) is that data ends up scattered across different tables. For example, consider the following example tables. Notice how a simple application is now split across these 4 tables - but we care more about who ordered what book, rather than the order IDs.

Customer

ID	Name	City
1	james	London
2	steve	London
3	kate	Manchester

Order

ID	CustomerID
1	1
2	2
3	3

OrderedItem

ID	CustomerID
1	1
1	2
2	1
3	3

Book

ID	Title	Author
1	Fahrenheit 451	Ray Bradbury
2	Animal Farm	George Orwell
3	Distributed Systems	van Steen & Tanenbaum

Joins are very common not only due to normalisation (with mostly **Foreign-Key joins**), but also due to the value produced by combining data. For example, we can find users that purchase the same product with joins, or finding what advertisements work (seeing which users search for a term within a timeframe after seeing an advert).

Joins are basically cross products with a selection **involving both inputs**. The variations on joins are based on how tuples without corresponding rows on either input are matched;

- **left join** $(R \overset{\text{L}}{\bowtie} S)$

  This returns all rows in R, even if no rows in S match (in this case it fills the columns of S with `NULL` values)

- **right join** $(R \overset{\text{R}}{\bowtie} S)$

  Same as above, but the inverse

- **full outer join** $(R \overset{\text{O}}{\bowtie} S)$

  Returns every row in R as well as every row in S (even if no rows are matching) - similarly will fill with `NULL` values if nothing is matching;

$$R \overset{\text{O}}{\bowtie} S \equiv (R \overset{\text{L}}{\bowtie} S) \cup (R \overset{\text{R}}{\bowtie} S)$$

An example of the matching function is as follows;

```
SELECT * FROM R JOIN S ON (R.r = S.s)
```

The matching function for joins does not have to be equality;

- **equi-join** (algorithmically equivalent to intersections) equality
- **inequality joins** inequality constraint (< or >)
- **anti-join** <> or !=
- **Theta joins** all other joins (difficult to optimise)

Join Algorithms

Some of the join algorithms are as follows;

- **nested loop join**

```
using Table = vector<vector<int>>;
Table left, right;
for (size_t i = 0; i < leftRelationSize; i++) {
    auto leftInput = left[i];
    for (size_t j = 0; j < rightRelationSize; j++) {
        auto rightInput = right[j];
        if (leftInput[leftAttribute] == rightInput[rightAttribute]) {
            writeToOutput({leftInput, rightInput});
        }
    }
}
```

This is a simple algorithm and is trivial to parallelise (since there are no dependent loop iterations). This also has sequential I/O, which is good for the buffer manager. The effort required however is;

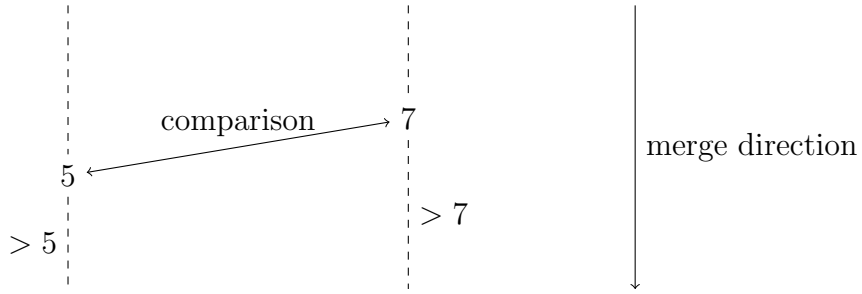
$$\Theta(|L| \times |R|) \text{ , reduced to } \Theta\left(\frac{|L| \times |R|}{2}\right) \text{ if uniqueness can be assumed}$$

- **sort-merge joins**

```
// this assumes values are unique and sorted
auto leftI = 0;
auto rightI = 0;
// keep iterating until either input is finished
while (leftI < leftInputSize && rightI < rightInputSize) {
    auto leftInput = left[leftI];
    auto rightInput = right[leftI];
    if (leftInput[leftAttribute] < rightInput[rightAttribute]) {
        leftI++;
    } else if (rightInput[rightAttribute] > leftInput[leftAttribute]) {
        rightI++;
    } else {
        writeToOutput({leftInput, rightInput});
        rightI++;
        leftI++;
    }
}
```

The idea of this is that we first sort both inputs and have cursors starting at the start of each input. If the value at the left pointer is smaller than the value at the right pointer, we increment the left pointer (and vice versa). If it's the same, then we can output a match.

Assuming (without loss of generality) the value of on the left is less than the value on the right. All values after the value on the right are greater than it (due to the sorted nature). Therefore no value after the value on the right can be a join partner to the value on the left, therefore we can advance the cursor on the left.



The effort required for this is as follows (the merge must go through each tuple on either side);

$$O(\underbrace{|L| \times \log |L|}_{O(\texttt{sort}(L))} + \underbrace{|R| + \log |R|}_{O(\texttt{sort}(R))} + \underbrace{|L| + |R|}_{O(\texttt{merge})})$$

This has sequential I/O in the merge phase, however it is tricky to parallelise. This also works for inequality joins, but we need to be more careful when advancing the cursors.

- **hash joins**

```
vector<optional<vector<int>>> hashTable; // optional since slots can be empty
int hash(int);
int nextSlot(int);

for (size_t i = 0; i < buildSide.size(); i++) {
    auto buildInput = buildSide[i];
    auto hashValue = hash(buildInput[buildAttribute]);

    // avoid overwriting a value
    while (hashTable[hashValue].hasValue) {
        hashValue = nextSlot(hashValue);
    }
    hashTable[hashValue] = buildInput;
}

for (size_t i = 0; i < probeSide.size(); i++) {
    auto probeInput = probeSide[i];
    auto hashValue = hash(probeInput[probeAttribute]);
    while (hashTable[hashValue].hasValue && hashTable[hashValue].value[
       buildAttribute] != probeInput[probeAttribute]) {
        hashValue = nextSlot(hashValue);
    }
    if (hashTable[hashValue].value[buildAttribute] == probeInput[
       probeAttribute]) {
        writeToOutput({hashTable[hashValue].value, probeInput})
    }
}
```

We need to first distinguish the **build-side** (side buffered in the hashtable) and **probe-side** (side used to look up tuples in the hashtable).

For this, we need to establish some requirements on the hash function. The requirements is that it is pure (and has no state), and need to know the output domain (the range of generated values)

in order to allocate the size. It's also ideal to have a contiguous output domain (without holes) and a uniform distribution (where all values are equally likely for consistent performance). Some common examples are;

- **MD5**
- **Modulo-Division** simplest (but input skew leads to output skew)
- **MurmurHash** one of fastest, decent hash-functions
- **CRC32** also has input skew issue, but has hardware support

Furthermore, we need to handle conflicts on hash collision. This needs to have some locality (if there is too much locality, there can be issues when inserting a lot of data into the same slot). Additionally, it needs to have no holes (we want to probe all slots to avoid memory waste). Some approaches are as follows;

- **linear probing**

  When a slot is filled, try the next one and continue doing so until a free slot is found - wrapping around to the start at the end of the buffer. This approach is simple and has great access locality. However it can lead to long probe-chains for adversarial input data (if we have input locality).
- **quadratic probing**

  This is similar to the above strategy, but we instead double the distance from the initial slot each time, first checking one that is a distance of 1 away, then 2, then 4, and so on, also wrapping at the end of the buffer. This is also simple, but only has good locality for the first few probes (and gets much worse). The first few probes are also likely to cause conflicts.
- **rehashing**

  We want to distribute probes uniformly, which can cause poor access locality but reduces conflicts. To do so, we can just use the hash function again. However, to ensure all slots are probed, we need to consider **cyclic groups**.

An example of this applied, with **modulo hashing** (with a constant factor of 10 for simplicity) and **linear probing** is as follows;

```
1 int hash(int v) { return v % 10; }
2 int probe(int v) { return (v + 1) % 10; }
3
4 buildSide = {1, 2, 7, 8, 12, 16, 17};
```

With this, we'd end up with the following slots in the hash table (- denotes unused);

```
(index 0) => [-, 1, 2, 12, -, -, 16, 7, 8, 17] <= (index 9)
```

Hash joins give us sequential I/O on inputs but pseudo-random access to the hash-table during build and probe. It's parallelisable over the values on the probe side, but parallelising the build is difficult. The effort required is;

$$\Theta(|\texttt{build}| + |\texttt{probe}|) \text{ (best case) and } O(|\texttt{build}| \times |\texttt{probe}|) \text{ (worst case)}$$

However, it's also important to consider that we typically want to store more than a single value, and often we may not have the uniqueness guarantee. Good hashing is still expensive and requires lots of CPU cycles (more expensive than multiple data accesses). Slots are often also allocated in buckets, allow for more than one tuple per slot. It's roughly equivalent to rounding every hash value down to a multiple of the bucket size. Sometimes buckets are implemented as linked lists (**bucket-chaining**, **open addressing**) which is a bad idea for lookup performance. Hashtables are also arrays, which occupy space (typically overallocated by a factor of two to reduce long

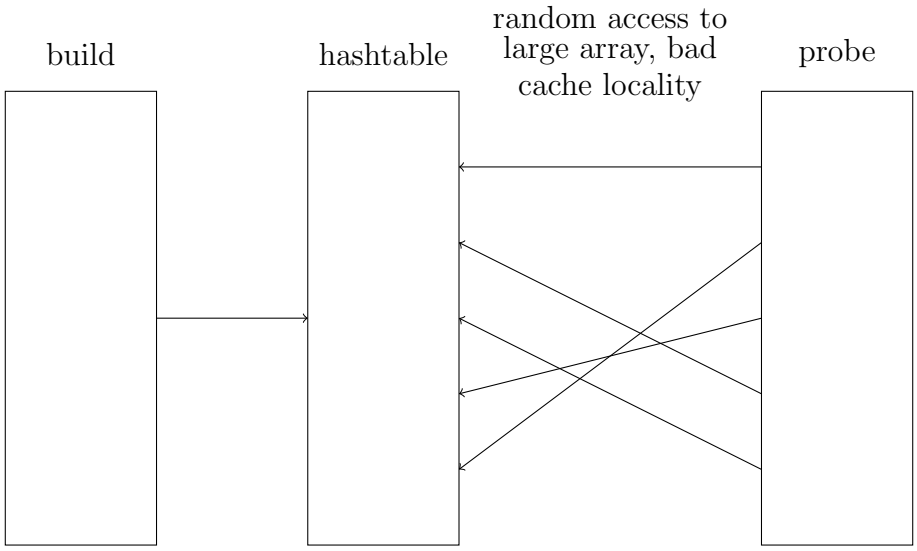
probe chains). They are also probed randomly in the probe phase, ideally we want to keep the hash table in memory / cache, but not to disk. **For this class**, if the hashtable doesn't fit, **every access has a constant penalty**. Generally we use hash joins when one relation is much smaller than the other.

Partitioning

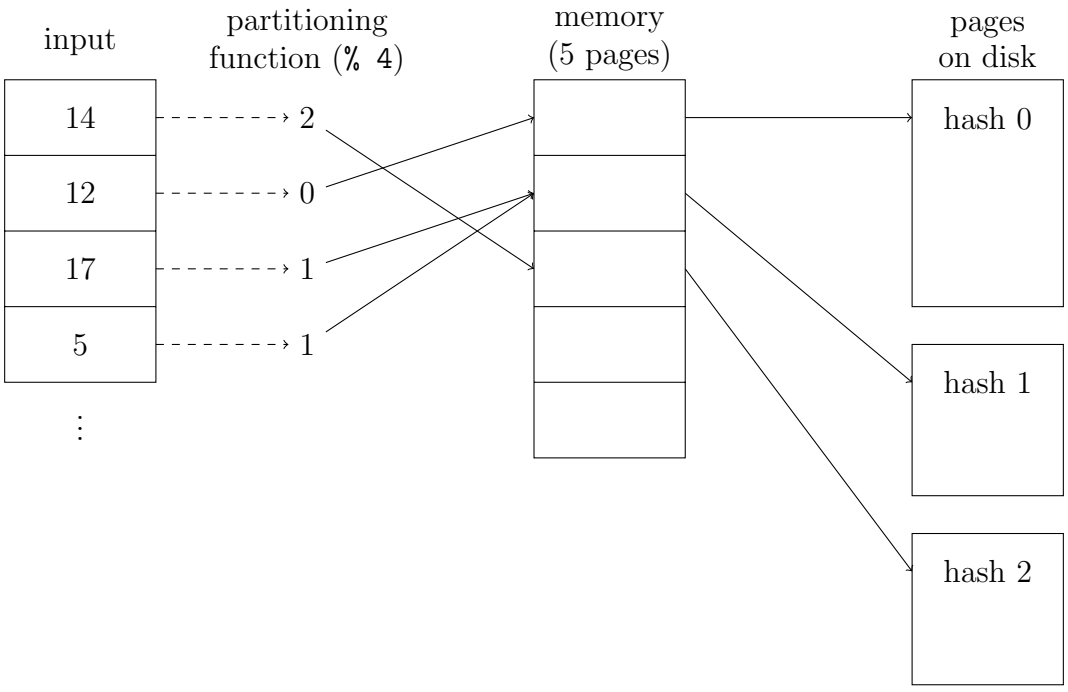
We have the fundamental premise that sequential access is much cheaper than random access, and the difference grows with the page size. If we assume a random value access cost c, the sequential value access cost is;

$$\frac{c}{\text{pagesize}_{\text{OS}}}$$

Assuming the hashtable doesn't fit in the page / cache, we will use the cost c quite frequently, and therefore the extra pass for partitioning, which is mostly sequential access might be worth doing. We can visualise thrashing as follows;



On the other hand, by using a partitioning function (which typically has a small range to fit stuff into memory);



The write from input to memory is done for every tuple, but the write from memory to the pages on the disk is only done when the page is **filled**, which doesn't waste bandwidth (and also lowers the

number of cs we use). As such, the probing becomes more involved, we first partition the probe in the same way, and we then only look at the partitioned hashtables, which gives us localised random access.

Another benefit of partitioning is that we can now parallelise the processing of each of the smaller joins, since we know they are disjoint.

Indexing

We should note that these algorithms are done in phases (build / probe and sort / merge). Typically the first phase, build or sort, is independent of the query phase, and can be done before any data is requested (such as when it is first loaded in). This is called indexing.

An index is a secondary storage, which is about replicating data. The replication is controlled by the DBMS, which means they can be created and destroyed without breaking the system. These replicas are semantically invisible to the user (doesn't change results). On the other hand, these replicas occupy space and need to be maintained under updates.

A primary index is used to store the tuples of a table, not a table, and therefore only one of these can be used per table. On the other hand, a secondary index stores pointers to the tuple of a table, hence we can have many of these in a table.

In SQL, an index can be created and destroyed as follows (however, it's unclear what type of index is created, nor do we have control over parameters);

```
1 CREATE INDEX index_name ON table_name (column1, column2, ...);
2 DROP INDEX index_name;
```

Some examples are as follows;

- **Hash-Indexing**

  The first step of a hash-join was building a hash-table, instead the hash-index is the same but persistent. In an unclustered hash-index, there is hash table which stores the key and position, and in the relation it is ordered by the position (and contains the actual data).

  The **ephemeral** hashtables we built for hash-joins were temporary, and we assume that no new tuples are added during the evaluation of the query, and we'd also know a rough amount of tuples that would end up in the table. However, all of this changes if the hash-table is persistent. Since persistent tables may grow arbitrarily large, we need to overallocate by a lot. If the fill-factor grows beyond x percent, we need to rebuild it with a larger overallocation - this can be expensive (and leads to load spikes on the inserts causing rebuilds). Similar, deletion can also be problematic.

  Previously, we used empty slots to mark an end of a probe-chain. However, a value must remain if something is deleted - we can either leave the value and mark it as deleted, or put the last value in the probe chain in the position. A deletion strategy (to delete key `k`) is as follows;

  - hash `k`, find `k`, and keep a pointer to `k`
  - continue probing until the end of the probe chain is found
  - if the value at the end of the chain has the same hash as `k`, move it into `k`'s slot
  - otherwise mark `k` as deleted (we can then fill this slot with the next value that hashes into the probe chain)

  An example is as follows. Note that 14 has the same hash as 23 (under modulo 9), but 14 has a different hash to 17.

before				
key	non-key			deleted
9	16	9	34	
27	5	61	45	
12	12	78	1	
23	84	17	69	
5	45	71	20	
17	9	42	83	
14	21	55	2	

after delete 23				
key	non-key			deleted
9	16	9	34	
27	5	61	45	
12	12	78	1	
14	21	55	2	
5	45	71	20	
17	9	42	83	

after delete 14				
key	non-key			deleted
9	16	9	34	
27	5	61	45	
12	12	78	1	
14	21	55	2	×
5	45	71	20	
17	9	42	83	

Similar to hash-joins, persistent hash-tables are good for hash-joins and aggregations (assuming they are built on the join / aggregation key columns). They also reduce the number of candidates for equality selections, but don't help on much else.

- **Bitmap-Index**

  It's first important to establish a **bitvector** is a sequence of 1-bit values indicating a boolean condition holding for the elements of a sequence of values. However, it's important to note that CPUs don't work with individual bits but rather words (let us assume 8 bit words, although in reality it's at least 32 bits);

$$\begin{aligned} BV_{==7}([4,7,11,7,7,11,4,7]) &= [0,1,0,1,1,0,0,1] \\ &= 128\cdot 0 + 64\cdot 1 + 32\cdot 0 + 16\cdot 1 + 8\cdot 1 + 4\cdot 0 + 2\cdot 0 + 1\cdot 1 \\ &= 89 \end{aligned}$$

  Bitmap indices are a collection of bitvectors on a column (one for each distinct value in that column). This is useful if there are few distinct values and these bitvectors are disjoint.

column	bitmap index		
	==5	==3	==9
5	1	0	0
3	0	1	0
9	0	0	1
9	0	0	1
9	0	0	1
5	1	0	0
3	0	1	0
9	0	0	1

  An implementation of this is as follows;

```
1  unsigned char** bitmaps;
2  void scanBitmap(byte* column, size_t inputSize, byte value) {
3
4      // finds specific bitmap for a certain value
5      unsigned char* scannedBitmap = bitmapForValue(bitmaps, value);
6
7      // iterate over bytes in bitmap
8      for (size_t i = 0; i < inputSize / 8; i++) {
9          // skip if none of the values are set
10         if (scannedBitmap[i] != 0) {
11             unsigned char bitmapMask = 128; // binary 10000000
12             for (size_t j = 0; j < 8; j++) {
```

```
13                  if ((bitmapMask & scannedBitmap[i]) && column[i * 8 + j] ==
                        value) {
14                      writeOutput(column[i * 8 + j]);
15                  }
16                  bitmapMask >>= 1;
17              }
18          }
19      }
20  }
```

This is useful as it reduces the bandwidth need for scanning a column, in the order of the size of the type of the column in bits (instead of loading a 32 bit value, we can just check 1 bit). Predicates can now also be combined using logical operators on the bitvectors. Additionally, arbitrary boolean conditions (such as ranges) can be indexed by some systems.

Binned bitmaps uses the idea of having n bitvectors, each with a predicate covering a different part of the value domain. For example if we assume our column type is a `byte`, we can have the following three bins;

- bin 1: 0 - 7
- bin 2: 8 - 20
- bin 3: 21 - 255

We need to ensure the conditions span the entire value domain. Also, the index cannot distinguish values in a bin (unless it only contains one value) - it can only produce **candidates**, some of which may be false positives.

Generally there are two strategies for binning;

- **equi-width**

  This is simple to configure - we can calculate our bin-width to be

  $$\frac{\max(\texttt{column}) - \min(\texttt{column})}{\texttt{numberOfBins}}$$

  This has limited use when indexing non-uniformly distributed data (since there can be a high frequency of false positives in a highly populated bin).
- **equi-height**

  This method is more resilient against non-uniformly distributed data - the false positive rate is value independent. We want to have the same number of counts per bin. The construction is more difficult however, as we need to determine quantiles and is usually performed on samples. If the distribution changes, it will need to be rebinned (which can be expensive).

Additionally, when we are binning, we may encounter many consecutive values which are equal. Each of these consecutive values can be replaced with the value (the Run) as well as the number of tuples (length) - this works very well on high-locality data. For example, we can transform `[0, 1, 1, 1, 1, 1, 0, 1]` to `[(0, 1), (1, 5), (0, 1), (1, 1)]`. However, this will now require a sequential scan, which can be fixed with **length prefix summing** (which stores the start of the run instead of the length); `[(0, 3), (1, 3), (0, 1), (1, 1)]` to `[(0, 0), (1, 3), (0, 6), (1, 7)]`. If we were to search for a value, we can just do a binary search instead of a sequential scan.

- **B-Trees**

  Databases are typically I/O bound (for disk), therefore we want to minimise page I/O operations. There are also many equality lookups, as well as many updates. Databases use high-fanout

trees to minimise page I/Os. We want the node of a tree to ideally be exactly the same size as a page (if we access a page, we might as well get the maximum amount of information out). Generally, a node in a B-Tree contains pairs (pivots) of keys and payloads, as well as child pointers to other nodes between the pairs (as well as before and after).

A B-Tree is defined as follows;

- a **balanced tree** with out-degree n (every node has $n-1$ keys)
- the **root** has at least one element
- each **non-root node** contains at least $\lfloor \frac{n-1}{2} \rfloor$ key / value pairs (at least half full)

Unlike other trees, B-Trees grow towards the root. In order to maintain balance, on an insertion the following steps apply;

- find the correct **leaf-node** to insert by walking the tree and inserting the value
- if this node overflows (the leaf was already full), split the node in two halves
  - take one pivot from the middle of the leaf and move it to the parent node
  - the two newly created nodes will become the left and right children of the pivot
- if the parent overflows, repeat the procedure on the parent
- if the parent is root, a new root node is introduced (therefore we add more roots, rather than add leaves)

For deletion, the following steps apply;

- find the value to delete
  - delete it if it is in a leaf node
  - otherwise, if it is in an internal node, keep a pointer to it, and replace it with the maximum leaf-node value from the left-child (removing the value from the leaf-node) - all modifications are on the leaf level
- if the affected leaf node underflows, we need to rebalance the tree bottom-up
  - try to obtain an element from a neighbouring node (to the right), move that up one level and make it the new splitting pivot, and take the old splitting pivot from the parent and put it into the leaf node
  - however, if that fails (the neighbour isn't more than half full), the nodes can be merged and the parent splitting key can be removed
  - if that causes an underflow, keep rebalancing upwards

While B-Trees can support ranges, it is complicated and requires going up and down the tree. This causes many node traversals, however node sizes are usually the same as page sizes, therefore traversals translate into page faults (which we want to minimise). Leaf pointers also aren't used, and most of the data lives in leaf nodes, therefore there is some space wasted.

- **B$^+$-Trees**

  The idea of this is to make range scans faster by keeping data only in the leaves, and linking leaf nodes to the next. Inner-node split values are replicas of leaf-node values. There is now more of a distinction between inner nodes and leaf nodes. The child pointers of the inner nodes are to other nodes, whereas the child pointers of leaf nodes (to the left) are now the payloads for a given key, and the right-most child pointer points to the next leaf. As such, the leaf nodes now form a sorted sequence.

  The balancing is largely the same as the balancing for regular B-Trees, however the deletion of an inner-node's split values imply a replacement with a new value from the leaf node.

- **Foreign-Key Index**

  In SQL, a foreign key is as follows; `ALTER TABLE Orders ADD FOREIGN KEY (BookID_index) REFERENCES Book(ID)`. FK constraints specify that for for every value that occurs in an attribute of a table, there is exactly one value in the PK column of another table. This constraint must be done by the DBMS, on an insertion or update, the DBMS needs to look up the PK value, and instead of storing the value, the DBMS could store a pointer to the referenced PK or tuple. We can think of this as equivalent to a pointer, and we can consider these as pre-calculated joins (which is often used since most joins are PK/FK joins from normalisation).

  Generally there are very few downsides - they cause insignificant work under updates, they do not cost much more space, and doesn't take more effort for query optimisation.