CT3532

DATABASE SYSTEMS II

Andreas Ó hAoḋa
University of Galway
2023-12-07

Contents

1 Introduction

1.1 Recommended Texts

- *Fundamentals of Database Systems* by Elmasri and Navathe 005.74 ELM
- *Database system concepts* by Silberschatz, A. 005.74 SIL

1.2 Assessment

Continuous Assessment accounts for 30% of the final grade, and the exam accounts for the remaining 70%. Plagiarism of assignments is not permitted - This is strictly enforced.

1.2.1 Assignments

2 Design

2.1 Re-cap

Normalisation can be used to develop a normalised relational schema given the universal relation, and verify the correctness of relational schema developed from conceptual design. We decompose relations such that it satisfies successively restrictive normal forms.

Desirable properties of a relational schema include:

- **Clear semantics of a relation:** The **semantics** of a relation refers to how the attributes grouped together in a relation are to be interpreted. If ER modelling is done carefully and the mapping is undertaken correctly, it is likely that the semantics of the resulting relation will be *clear*. One should try to design a relation so that it is easy to explain its meaning.
- **Reducing the number of redundant values in tuples:** The presence of redundancy leads to waste of storage space and potential for anomalies (deletion, update, insertion). One should try to design relations so that no anomalies may occur. If an anomaly can occur, it should be noted. Normalisation will remove many of the potential anomalies.
- **Reducing the number of null values in tuples:** Having null values is often necessary, but it can create problems, such as:
  - Wasted space.
  - Different interpretations, i.e.: attribute does not apply to this tuple, attribute value is unknown, attribute value is known but absent, etc.
- **Disallowing the possibility of generating spurious tuples:** If a relation R is decomposed into R_1 & R_2 and connected via a primary key – foreign key pair, then performing an equi-join between R_1 & R_2 on the involved keys should not produce tuples that were not in the original relation R.

More formally, we typically have a relation R and a set of functional dependencies F, defined over R. We wish to create a decomposition $D = \{R_1, R_2, \ldots, R_n\}$ We wish to guarantee certain properties of this decomposition. We require that all attributes in the original R be maintained in the decomposition, *id est*: $R = R_1 \cup R_2 \cup \cdots \cup R_n$

- A relation is said to be in the **First Normal Form (1NF)** if there are no repeating fields.
- A relation is said to be in the **Second Normal Form (2NF)** if it is in 1NF and if every non-prime attribute is fully functionally dependent on the key.
- A relation is said to be in the **Third Normal Form (3NF)** if it is in 2NF and if no non-prime attribute is transitively dependent on the key.
- A relation is said to be in the **Boyce-Codd Normal Form (BCNF)** if the relation is in 3NF and if every determinant is a candidate key.

2.1.1 Example

StudentNo	Major	Advisor
123	I.T.	Smith
123	Econ	Murphy
444	Biol.	O' Reilly
617	I.T.	Jones
829	I.T.	Smith

Table 1: Sample Data

Constraints:

- A student may have more than one major.
- For each major, a student can have only one advisor.
- Each major can have several advisors.
- Each advisor advises one major.
- Each advisor can advise several students.

Functional dependencies:

- {StudentNo, Major} $\rightarrow$ {Advisor}
- {Advisor} $\rightarrow$ {Major}

An update anomaly may exist: If student 44 changes major, we lose information that O' Reilly supervises Biology. To solve this, we can decompose the tables so as to satisfy BCNF:

- TAKES: StudentNo, Advisor
- ADVISES: Advisor, Major

2.1.2 General Rule

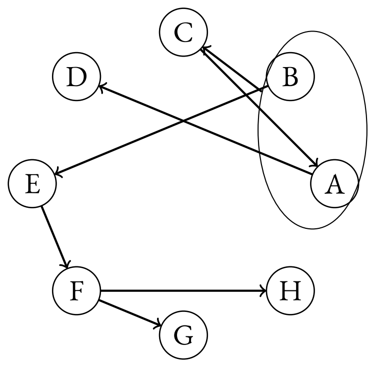
Consider a relation R with functional dependencies F. If $X \rightarrow Y$ violates BCNF, decompose R into:

- $\{R - Y\}$
- $\{XY\}$

2.1.3 Exercise

Let $R = \{A, B, C, D, E, F, G, H\}$. The functional dependencies defined over R are:

- $A \rightarrow D$
- $B \rightarrow E$
- $E \rightarrow F$
- $F \rightarrow G$
- $F \rightarrow H$
- $\{A, B\} \rightarrow C$
- $C \rightarrow A$



Decompose R such that the BCNF is satisfied.

3 Design by Synthesis

3.1 Background

Typically, we have the relation R and a set of functional dependencies F. We wish to create a decomposition $D = R_1, R_2, \ldots, R_m$. Clearly, all attributes of R must occur in at least one schema R_i, *id est*: $U_{i=1}^{m} R_i = R$. This is known as the **attribute preservation** constraint.

A **functional dependency** is a constraint between two sets of attributes. A functional dependency $X \rightarrow Y$ exists for all tuples t_1 & t_2 if $t_1[X] = t_2[X]$, then t_2 if $t_1[Y] = t_2[Y]$. We usually only specify the obvious functional dependencies, there may be many more. Given a set of functional dependencies F, the **closure of *F***, denoted F^+, refers to all dependencies that can be derived from F.

3.1.1 Armstrong's Axioms

Armstrong's Axioms are a set of inference rules that allow us to deduce all functional dependencies from a given initial set. They are:

- **Reflexivity:** if $X \supseteq Y$, then $X \rightarrow Y$.
- **Augmentation:** if $X \rightarrow Y$, then $XZ \rightarrow YZ$.
- **Transitivity:** if $X \rightarrow Y, Y \rightarrow Z$, then $X \rightarrow Z$.
- **Projectivity:** if $X \rightarrow YZ$, then $X \rightarrow Z$.
- **Additivity:** if $X \rightarrow Y, X \rightarrow Z$, then $X \rightarrow YZ$.
- **Pseudo-transitivity:** if $X \rightarrow Y, WY \rightarrow Z$, then $WX \rightarrow Z$.

The first three rules have been shown to be **sound** & **complete**:

- **Sound:** Given a set F on a relation R, any dependency we can infer from F using the first three rules holds for every state of r of R that satisfies the dependencies in F.
- **Complete:** We can use the first three rules repeatedly to infer all possible dependencies that can be inferred from F.

For any sets of attributes A, we can infer A^+, the set of attributes that are functionally determined by A given a set of functional dependencies.

3.1.2 Cover Sets

A set of functional dependencies F, **covers** a set of functional dependencies E if every functional dependency in E is in F^+.

Two sets of functional dependencies E & F are equivalent if $E^+ = F^+$. We can check if F covers E by calculating A^+ with respect to F for each functional dependency $A \rightarrow B$ and then checking that A^+ includes all the attributes of B.

A set of functional dependencies, F< is **minimal** if:

- Every functional dependency in F has a single attribute for its right-hand side.
- We cannot remove any dependency from F and maintain a set of dependencies equivalent to F.
- We cannot replace any dependency $X \rightarrow A$ with a dependency $Y \rightarrow A$ where $Y \subset X$, and still maintain a set of dependencies equivalent to F.

All functional dependencies $X \rightarrow Y$ specified in F should exist in one of the schema R_i, or should be inferable from the dependencies in R_i; this is known as the **dependency preservation** constraint. Each functional dependency specifies some constraint; if the dependency is absent, then some desired constraint is also absent. If a functional dependency is absent, then we must enforce the constraint in some other manner; this can be inefficient.

Given F & R, the **projection** of F on R_i, denoted $\pi_{R_i}(F)$ where R_i is a subset of R, is the set $X \rightarrow Y$ in F^+ such that attributes $X \cup Y \in R_i$. A decomposition of R is dependency-preserving if:

$$((\pi_{R_1}(F)) \cup \cdots \cup (\pi_{R_m}(F)))^+ = F^+$$

Theorem: It is always possible to find a decomposition D with respect to F such that:

1. The decomposition is dependency-preserving.
2. All R_i in D are in 3NF.

We can always guarantee a dependency-preserving decomposition to 3NF. **Algorithm:**

1. Find a minimal cover set G for F.
2. For each left-hand side X of a functional dependency in G, create a relation $X \cup A_1 \cup A_2 \dots A_m$ in D, where $X \rightarrow A_1, X \rightarrow A_2, \dots$ are the only dependencies in G with X as a left-hand side.
3. Group any remaining attributes into a single relation.

3.1.3 Lossless Joins

Consider the following relation:

- EMPPROJ: $\underline{\text{ssn, pnumber}}$, hours, ename, pname, plocation

and its decomposition to:

- EMPPROJ1: $\underline{\text{ename, plocation}}$
- EMPLOCAN: $\underline{\text{ssn, pno}}$, hrs, pname, plocation

If we perform a natural join on these relations, we may generate spurious tuples. When a natural join is issued against relations, no spurious tuples should be generated. A decomposition $D = \{R_1, R_2, \dots R_n\}$ of R has the **lossless join** (or non-additive join) property with regards to F on R if every instance r of the following holds:

$$\bowtie (\pi_{R_1}(r), \dots \pi_{R_m}(r)) = r$$

We can automate a procedure for testing for the lossless property. We can also automate the decomposition of R into $R_1, \dots R_m$ such that is possesses the lossless join property.

A decomposition $D = \{R_1, R_2\}$ has the lossless property if & only if:

- The functional dependency $(R_1 \cap R_2) \to \{R_1 - R_2\}$ is in F^+, *or*
- The functional dependency $(R_1 \cap R_2) \to \{R_2 - R_1\}$ is in F^+.

Furthermore, if a decomposition has the lossless property and we decompose one of R_i such that this also is a lossless decomposition, then replacing that decomposition of R_i in the original decomposition will result in a lossless decomposition.

Algorithm: To decompose to BCNF:

1. Let $D = R$.
2. While there is a schema B in D that violates BCNF, do:
   (a) Find the functional dependency $(X \to Y)$ that violates BCNF.
   (b) Replace B with $(B - Y)$ & $(X \cup Y)$.

This guarantees a decomposition such that all attributes are preserved, the lossless join property is enforced, and all R_i are in BCNF. It is not always possible to decompose R into a set of R_i such that all R_i satisfy BCNF and properties of lossless joins & dependency preservation are maintained. We can guarantee a decomposition such that:

- All attributes are preserved.
- All relations are in 3NF.
- All functional dependencies are maintained.
- The lossless join property is maintained.

Algorithm: Finding a key for a relation schema R.

1. Set $K := R$.
2. For each attribute $A \in K$, compute $(K - A)^+$ with respect to the set of functional dependencies. If $(K - A)^+$ contains all the attributes in R, the set $K := K - \{A\}$.

Given a set of functional dependencies F, we can develop a minimal cover set. Using this, we can decompose R into a set of relations such that all attributes are preserved, all functional dependencies are preserved, the decomposition has the lossless join property, and all relations are in 3NF. The advantages of this are that it provides a good database design and can be automated. The primary disadvantage is that often, numerous good designs are possible.

4 B Trees & B+ Trees

4.1 Generalised Search Tree

In a **Generalised Search Tree**, each node has the format $P_1, K_1, P_2, K_2, \dots, P_{n-1}, K_{n-1}, P_n$ where P_i is a **tree value** and K_i is a **search value**. Hence, the number of values per node depends on the size of the key field, block size, & block pointer size. The following constraints hold:

- $K_1 < K_2 < \dots < K_{n-1} < K_n$.
- For all values x in a sub-tree pointed to P_i, $K_{i-1} < x < K_i$.
- The number of tree pointers per node is known as the **order** or ρ of the tree.

4.1.1 Efficiency

For a generalised search tree: $T(N) = O(\log(N))$, assuming a balanced tree. In order to guarantee this efficiency in searching & other operations, we need techniques to ensure that the tree is always balanced.

4.2 B Trees

A **B tree** is a balanced generalised search tree. B trees can be viewed as a dynamic multi-level index. The properties of a search tree still hold and the algorithms for insertion & deletion of values are modified in order. The node structure contains a record pointer for each key value. The node structure is as follows:

$$P_1 < K_1, Pr_1 > P_2 < K_2, Pr2 > \dots P_{n-1} < K_{n-1}, Pr_{n-1} > P_n$$

4.2.1 Example

Consider a B Tree of order 3 (two values and three tree pointers per node/block). Insert records with key values: 10, 6, 8, 14, 4, 16, 19, 11, 21.

4.2.2 Algorithm to Insert a Value into a B Tree

1. Find the appropriate leaf-level node to insert a value.
2. If space remains in the leaf-level node, then insert the new value in the correct location.
3. If no space remains, we need to deal with collisions.

4.2.3 Dealing with Collisions

1. Split node into left & right nodes.
2. Propagate the middle value up a level and place its value in a node there. Note that this propagation may cause further propagations and even the creation of a new root node.
3. Place the values less than the middle value in the left node.
4. Place the values greater than the middle value in the right node.

This maintains the balanced nature of the tree, and $O(\log_\rho(N))$ for search, insertion, & deletion. However, there is always potential for unused space in the tree. Empirical analysis has shown that B trees remain 69% full given random insertions & deletions.

4.2.4 Exercise

Can you define an algorithm for deletion (at a high level)? How much work is needed in the various cases (best, average, worst)?

4.3 B+ Trees

The most commonly used index type is the **B+ tree** – a dynamic, multi-level index. B+ trees differ from B trees in terms of structure, and have slightly more complicated insertion & deletion algorithms. B+ trees offer increased efficiency over a B Tree and ensures a higher order ρ.

4.3.1 Node Structure

B+ trees have two different node structures: internal nodes & leaf-level nodes. The internal node structure is:

$$P_1, K_1, P_2, K_2, \dots P_{n-1}, K_{n-1}, P_n$$

All record pointers are maintained at the leaf level in a B+ tree. There are no record pointers in the internal nodes. B+ trees have less information per record and hence more search values per node.

One tree pointer is maintained at each leaf-level node, which points to the next leaf-level node. Note that there is only one tree pointer per node at the leaf level. Each leaf-level node's structure is:

$$K_1, Pr_1, K_2, PR_2, \dots K_m, Pr_m, P_{\text{next}}$$

4.3.2 Example

Let $B = 512$, $P = 6$, $K = 10$. Assume 30,000 records as before. Assume that the tree is 69% full. How many blocks will the tree require? How many block accesses will a search require?

4.3.3 Example

A tree of order ρ has at most $\rho - 1$ search values per node. For a B+ tree, there are two types of tree nodes; hence there are two different orders: ρ & ρ_{leaf}. To calculate ρ_{leaf}:

$$|P| + (\rho_{\text{leaf}})(|K| + |Pr|) \leq B$$

$$\rightarrow 17(\rho_{\text{leaf}}) \leq 506$$

$$\rho_{\text{leaf}} = 29$$

Given a fill factor of 69%:

- Each internal node will have, on average, 22 pointers.
- Each leaf-level node will have, on average, 20 pointers.

- Root: 1 node, 21 entries, 22 pointers.
- Level 1: 22 nodes, 462 entries, 484 pointers.
- Level 2: 484 nodes, ..., etc.
- Leaf Level: ...

Hence, 4 levels are sufficient. The number of block accesses $= 4 + 1$. The number of blocks is $1 + 22 + 484 + \dots$

5 Hash Tables

5.1 Introduction

Can we improve upon logarithmic searching? **Hashing** is a technique that attempts to provide constant time for searching & insertion, i.e. $O(K)$. The basic idea for searching & insertion is to apply a hash function to the search field of the record; the return value of the hash function is used to reference a location in the hash table.

Care should be taken in designing a hash function. We usually require a **fair** hash function. This is difficult to guarantee if there is no or limited information available about the type of data to be stored. Often, heuristics can be used if domain knowledge is available. We can have both internal (i.e., some data structure in memory) or external hashing (i.e., to file locations). We must consider the size of the original table or file.

5.2 Approaches

1. Create a hash table containing N addressable "slots" which each can contain one record.

2. Create a hash function that returns a value to be used in insertion & searching. The value returned by the hash function must be in the correct range, i.e. the address space of the hash table

If the range of the keys is that of the address space of the table, we can guarantee constant-time lookup. However, this is usually not the case as the address space of the table is much smaller than that of the search field.

With numeric keys, we can use modulo-division or truncation for hashing. With character keys, we must first convert to an integer value: this can be achieved by multiplying the ASCII code of the characters together and then applying modulo-division. However, we cannot guarantee constant-time performance as collisions will occur, i.e. two records with different search values being hashed to the same location in the table; we require a collision resolution policy. Efficiency then will depend on the number of collisions. The number of collisions depends primarily on the load factor λ of the file:

$$\lambda = \frac{\text{Number of records}}{\text{Number of slots}}$$

5.2.1 Collision Resolution Policies

- **Chaining:** if a location is full, add the item to a linked list. Performance degrades if the load factor is high. The lookup time is on average $1 + \lambda$.

- **Linear probing:** if a location is full, then check in a linear manner for the next free space. This can degrade to a linear scan. The performance, if successful is $0.5(1 + \frac{1}{1-\lambda})$ and if unsuccessful is $0.5 + (1 + \frac{1}{1+\lambda}^2)$. One big disadvantage of this approach is that it leads to the formation of clusters.

- **Quadratic probing:** if a location is full, check the location $x + 1$, location $x + 4$, location $(x + n)^2$. This results in less clustering.

- **Double hashing:** if location x is occupied, then apply a second hash function. This can help guarantee an even distribution (a fairer hash function).

5.3 Dynamic Hashing

The cases that we've considered thus far deal with the idea of a **fixed hash table**: this is referred to as **static hashing**. Problems arise if the database grows larger than planned: too many overflow buckets and performance degrades. A more suitable approach is **dynamic hashing**, wherein the table or file can be re-sized as needed.

5.3.1 General Approach

1. Use a family of hash functions h_0, h_1, h_2, etc. h_{i+1} is a refinement of h_i. E.g., $K \text{mod} 2^i$.

2. Develop a base hash function that maps the key to a positive integer.

3. Use $h_0(x) = x \text{mod} 2^b$ for a chosen b. There will be 2^b buckets initially. We can effectively double the size of the table by incrementing b.

We only double the number of buckets when re-organising conceptually; we do not actually double the number of buckets in practice as it may not be needed.

5.4 Dynamic Hashing Approaches

Common dynamic hashing approaches include extendible hashing & linear hashing. **Extendible hashing** involves re-organising the buckets when & where needed, whereas **linear hashing** involves re-organising buckets when but not where needed.

5.4.1 Extendible Hashing

- When a bucket overflows, split that bucket in two. A directory is used to achieve this conceptual doubling.
- If a collision or overflow occurs, we don't re-organise the file by doubling the number of buckets, as this would be too expensive. Instead, we maintain a directory of pointers to buckets, we can effectively double the number of buckets by doubling the directory, splitting just the bucket that overflowed. Doubling the directory is much cheaper than doubling the file, as the directory is much smaller than the file,
- On overflow, we split the bucket by allocating a new bucket and redistributing its contents. We double the directory size if necessary.

We maintain a **local depth** for each bucket, effectively the number of bits needed to hash an item here. We also maintain a **global depth** for the directory which is the number of bits used in indexing items. We can use these values to determine when to split the directory.

- If overflow occurs in a bucket where the local depth = global depth, then split the bucket, redistribute its contents, and double the directory.
- If overflow occurs in a bucket where the local depth < global depth, then split the bucket, redistribute its contents, and increase the local depth.

If the directory can fit in the memory, then the retrieval for point queries can be achieved with one disk read.

5.5 Linear Hashing

Linear hashing is another approach to indexing to a dynamic file. It is similar to dynamic hashing in that a family of hash functions are used ($h = K \bmod 2^i$), but differs in that no index is needed. Initially, we create a file of M buckets; $K \bmod M^1$ is a suitable hash function. We will use a family of such functions $K \bmod (2^i \times M), i = 0$ initially. We can view the hashing as comprising of a sequence of phases: for phase j, the hash functions $K \bmod 2^i \times M$ & $K \bmod 2^{j+1} \times M$ are used.

We **split a bucket** by redistributing the records into two buckets: the original one & a new one. In phase j, to determine which ones go into the original while the others go into the new one, we use $H_{j+1}(K) = K \bmod 2^{j+1} \times M$ to calculate their address. Irrespective of the bucket which causes the overflow, we always split the next bucket in a **linear order**. We begin with bucket 0, and keep track of which bucket to split next p. At the end of a phase when p is equal to the number of buckets present at the start of the phrase, we reset p and a new phase begins (j is incremented).

6 Joins

Many approaches & algorithms can be used to do **joins**.

6.1 Nested Loop Join

To perform the join $r \bowtie s$:

```
for each tuple t_r in r do:
    for each tuple t_s in s do:
        if t_r and t_ satisfy join condition:
            add(t_r, t_s) to result
    end
end
```

This is an expensive approach; every pair of tuples is checked to see if they satisfy the join condition. If one of the relations fits in memory, it is beneficial to use this in the inner loop (known as the **inner relation**).

6.2 Block nested Loop Join

The **block nested loop join** is a variation on the nested loop join that increases efficiency by reducing the number of block accesses.

```
for each block B_r in r do:
    for each block B_s in s do:
        for each tuple t_r in B_r do:
            for each tuple t_s in B_s do:
                if t_r and t_s satisfy join condition:
                    add (t_r, t_s) to result
            end
        end
    end
end
```

6.3 Indexed Nested Loop Join

If there is an index available for the inner table in a nested loop join, we can replace file scans with index accesses.

6.4 Merge Join

If both relations are sorted on the joining attribute, then we can merge the relations. The technique is identical to merging two sorted lists (such as in the "merge"' step of a Merge-Sort algorithm). Merge joins are much more efficient than a nested join. They can also be computed for relations that are not ordered on a joining attribute, but have indexes on the joining attribute.

6.5 Hash Joins

1. Create a hashing function which maps the join attribute(s) to partitions in a range $1..N$.
2. For all tuples in r, hash the tuples to H_{ri}.
3. For all tuples in s, hash the tuples to H_{si}.
4. For $i = 1$ to N, join the partitions $H_{ri} = H_{si}$.

7 Sorting

Sorting is a very important operation because it is used if a query specifies `ORDER BY` and is used prior to relational operators (e.g. Join) to allow more efficient processing of the operation. We can sort a relation in two ways:

- **Physically:** actual order of tuples re-arranged on the disk.
- **Logically:** build an index and sort index entries.

When the relation to be sorted fits in the memory, we can use standard sorting techniques (such as Quicksort). However, when the relation doesn't fit in memory, we have to use other approaches such as the **external sort merge**, which is essentially an N-way merge, an extension of the idea in the merge step of the merge sort algorithm.

```
i := 0;
M = number of page frames in main memory buffer

repeat
    read M blocks of the relation
    sort M blocks in memory
    write sorted data to file R_i
```

```
until end of relation

read first block of each R_i into memory
repeat
    choose first (in sort order) from pages
    write tuple to output
    remove tuple from buffer
    if any buffer R_i empty and not eof(R_i)
        read next block from R_i into memory
    until all pages empty
```

8 Parallel Databases

Characteristics of **Parallel databases** include:

- Increased transaction requirements.
- Increased volumes of data, particularly in data-warehousing.
- Many queries lend themselves easily to parallel execution.
- Can reduce the time required to retrieve relations from disk by partitioning relations onto a set of disks.
- Horizontal partitioning is usually used. Subsets of a relation are sent to different disks.

8.1 Query Types

Common types of queries include:

- **Batch processing:** scanning an entire relation.
- **Point-Queries:** return all tuples that match some value.
- **Range-Queries:** return all tuples with some value in some range.

8.2 Partitioning Approaches

8.2.1 Round Robin

With **Round Robin**, the relation is scanned in order. Assuming n disks, the i^{th} relation is sent to disk $D_i \text{mod} n$. Round Robin guarantees an even distribution

Round Robin is useful for batch processing, but is not very suitable for either point or range queries as all disks have to be accessed.

8.2.2 Hash Partitioning

In **hash partitioning**, we choose attributes to act as partitioning attributes. We define a hash function with range $0 \ldots n-1$, assuming n disks. Each tuple is placed according to the result of the hash function.

Hash partitioning is very useful if a point query is based on a partitioning attribute. It is usually useful for batch querying if a fair hash function is used, but is poor for range querying.

8.2.3 Range Partitioning

In **range partitioning**, a partitioning attribute is first chosen. The partitioning vector is defined as $< v_0, v_1, \dots, v_{n-2} >$. Tuples are placed according to the value of the partitioning attribute. If $t_{\text{partitioning attribute}} < v_0$, then we place tuple t on disk D_0.

Range partitioning is useful for both point & range querying, but can lead to inefficiency in range querying if many tuples satisfy the condition.

8.3 Types of Parallelism

8.3.1 Inter-Query Parallelism

In **inter-query parallelism**, different transactions run in parallel on different processors, thus increasing the transaction throughput, although the times for individual queries remain the same. Inter-query parallelism is the easiest for of parallelism to implement.

8.3.2 Intra-Query Parallelism

Intra-query parallelism allows us to run a single query in parallel on multiple processors (& disks), which can speed up the running time of a query. Parallel execution can be achieved by parallelising individual components, which is called **intra-operation parallelism**. Parallel execution can also be achieved by evaluating portions of the query in parallel, which is called **inter-operation parallelism**. Both of these types can be combined.