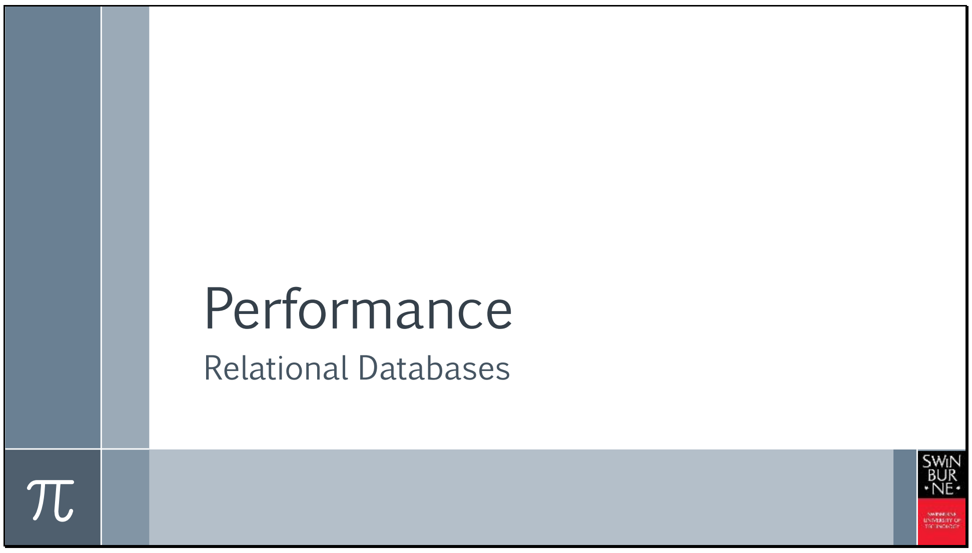
Slide 1

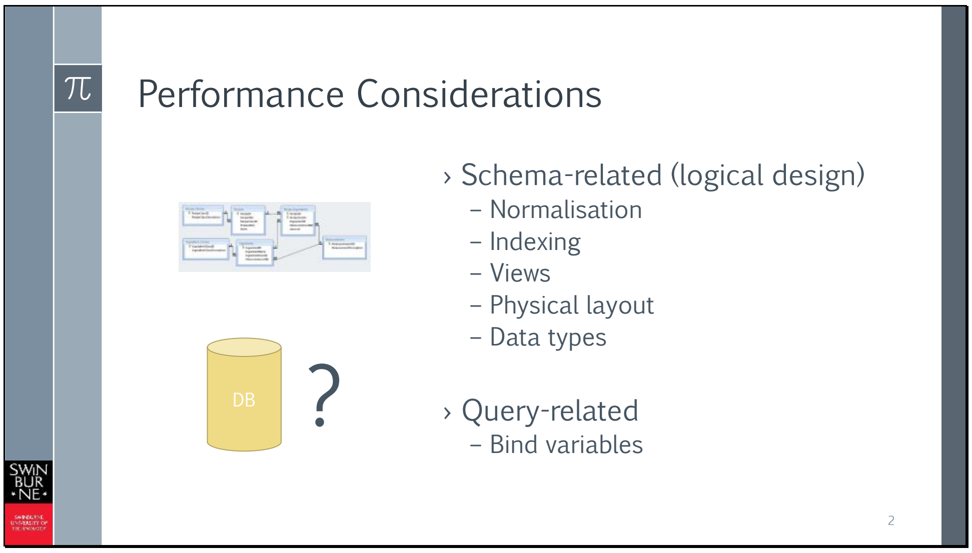


Performance

Relational Databases

If your database is being accessed by thousands of users a day, performance is critical. Users cannot wait five minutes for a web page to build.
These days developers often consider NoSQL databases for storage. One reason is that they are considered faster. We are going to discuss NoSQL data storage in the next module.
But NoSQL databases are not the answer to every situation. In this module we discuss how to make the right decisions in relational data management that ensures the performance is as good as it can be.

Slide 2



Performance Considerations

- Schema-related (logical design)
  - Normalisation
  - Indexing
  - Views
  - Physical layout
  - Data types
- Query-related
  - Bind variables

2

When performance becomes an issue, we usually become aware of the problem when our DML statements take a long time to execute. The most affected DML statements are SELECT queries, but UPDATEs, INSERTs and DELETEs can also become slow.
There are many reasons why the database can become unresponsive. We may have a distributed database with some parts being inaccessible temporarily, or we may have an unusual spike in the demand, or we may not have allocated enough memory space for the DBMS to work in. Many of these issues should be addressed by a database administrator, and we won't discuss them here.
In this module, we look at issues the programmer or database designer can and should address.

Performance is not a consideration to leave till later. From the issues listed you can see that most of the factors that influence the performance are schema-related. The logical design of the schema means what relations the schema contains, what attributes the relations have and how the relationships between the relations have been designed. Another aspect is the physical design – how the items are laid out in permanent storage. Relational databases offer a few ways for the user to specify how the data is laid out on the disk. For example, when you create a table, you can specify how much space this table can take on disk. If the table needs more space than you allocate, the table later has to expand to other areas of the disk, leading to fragmentation. We won't have time here to discuss this topic further.

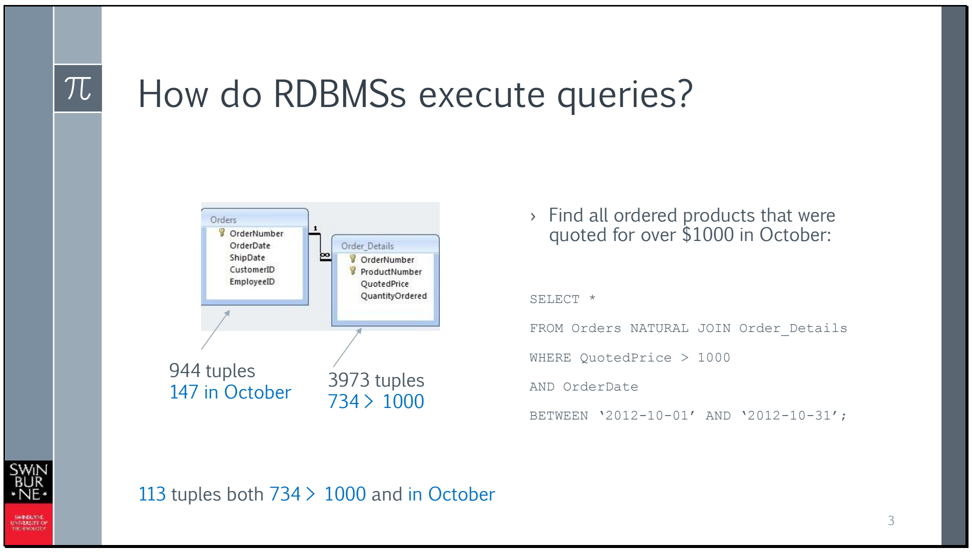
The logical design affects the time queries take. The relational model emphasises the importance of normalising the schema to prevent redundancies. When we have a perfectly normalised database, the data is separated into many different relations. This means that the queries often have to perform JOINs between the tables. JOINs are expensive – they are, in effect, search operations.

SELECT operations have to find items in a table. The larger the table, the more data has to be examined for a match. Indexes can help solve this problem by changing the way the tuples are accessed.

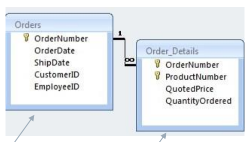
Views are stored SELECT statements. When SELECT statements are stored, they are pre-optimised. This means executing them takes less time than executing new SELECT statements the DBMS has never seen before.
The choice of data type can also add to the fragmentation of the database. More about this later.

Bind variables are an important concept for programmers. Using them when needed will improve the performance of queries, because bind variables also let the DBMS skip query optimisation steps.

Slide 3



How do RDBMSs execute queries?



944 tuples
147 in October

3973 tuples
734 > 1000

› Find all ordered products that were quoted for over $1000 in October:

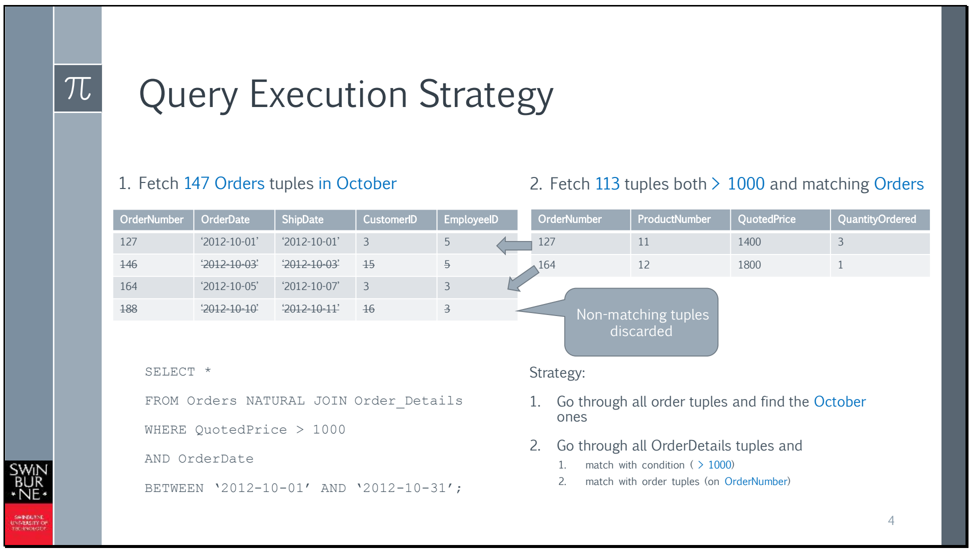
```
SELECT *
FROM Orders NATURAL JOIN Order_Details
WHERE QuotedPrice > 1000
AND OrderDate
BETWEEN '2012-10-01' AND '2012-10-31';
```

113 tuples both 734 > 1000 and in October

3

How do RDBMSs execute queries?
Suppose we want to find all products that were sold in October and most more than $1000.
The OrderDate is in the Orders table and the QuotedPrice is in the order details table. We have to join the tables to find the items that match both conditions. Natural join joins tables using a column that has the same name in both tables, here it is order number.
Neither the OrderDate nor the QuotedPrice column are indexed.
Because there are no indexes, the only possibility for the DBMS to find the matching tuples is by going through the relation sequentially – to look at every one of the 944 order rows to see if it is an October one, and every one of the 3973 details rows to see if the quoted price is over 1000. When we have all these rows, there will be order rows which do not have a related details row with a quoted price of over 1000, and there will be order details rows which are not from the month of October. We have to match the rows by OrderNumber. In set theory, this is known as an intersection. The tuples that are outside the intersection are discarded.

Slide 4



Query Execution Strategy

1. Fetch 147 Orders tuples in October

OrderNumber	OrderDate	ShipDate	CustomerID	EmployeeID
127	'2012-10-01'	'2012-10-01'	3	5
~~146~~	~~'2012-10-03'~~	~~'2012-10-03'~~	~~15~~	~~5~~
164	'2012-10-05'	'2012-10-07'	3	3
~~188~~	~~'2012-10-10'~~	~~'2012-10-11'~~	~~16~~	~~3~~

2. Fetch 113 tuples both > 1000 and matching Orders

OrderNumber	ProductNumber	QuotedPrice	QuantityOrdered
127	11	1400	3
164	12	1800	1

Non-matching tuples discarded

```
SELECT *

FROM Orders NATURAL JOIN Order_Details

WHERE QuotedPrice > 1000

AND OrderDate

BETWEEN '2012-10-01' AND '2012-10-31';
```

Strategy:

1. Go through all order tuples and find the October ones
2. Go through all OrderDetails tuples and
   1. match with condition (> 1000)
   2. match with order tuples (on OrderNumber)

4

There are a few strategies a DBMS could use to execute this query, and there are differences in execution between different database products.

Since we have restrictions on both tables, looking through each table and filtering the rows that match the restriction are unavoidable steps. But the order in which this happens depends on the strategy of the DBMS.
It could first combine all tuples of both Orders and OrderDetails and then apply the filters – the search conditions - on the combined rows. This would mean that we need to bring all of the Orders and all of the OrderDetails rows into memory first. Not a very fast strategy.

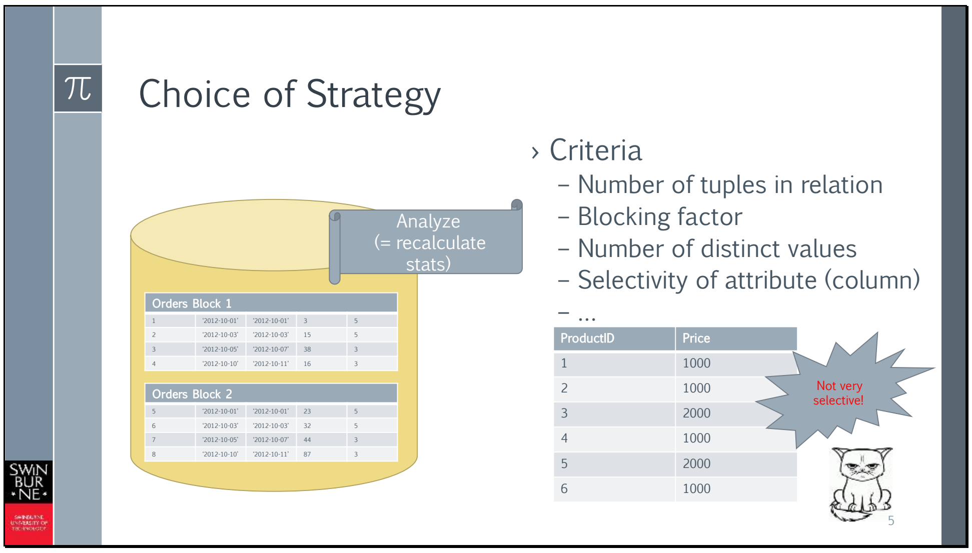
The DBMS could first fetch all rows from the Orders table that match the October restriction, then match them with the details rows and finally eliminate the ones with a quoted price of less than 1000. Because there are more OrderDetails rows than Orders rows, restricting the OrderDetails query as much as possible is a good strategy to reduce the number of rows to bring.
When we bring the Orders rows first, we can tell from the foreign – primary key match which rows of the OrderDetails we don't need. The restriction on the OrderDetails (the price) helps reduce the numbers further.

Not surprisingly, this is what MySQL does: It finds all the October Orders rows, then filters the order details rows according to two conditions at the same time: whether they match the chosen Order rows and whether they have a quoted price of over 1000.

Applying both restrictions means each row in the OrderDetails table has to be looked at only once.

I

Slide 5



Choice of Strategy

Analyze
(= recalculate stats)
Not very selective!
5

Orders Block 1				
1	'2012-10-01'	'2012-10-01'	3	5
2	'2012-10-03'	'2012-10-03'	15	5
3	'2012-10-05'	'2012-10-07'	38	3
4	'2012-10-10'	'2012-10-11'	16	3

Orders Block 2				
5	'2012-10-01'	'2012-10-01'	23	5
6	'2012-10-03'	'2012-10-03'	32	5
7	'2012-10-05'	'2012-10-07'	44	3
8	'2012-10-10'	'2012-10-11'	87	3

› Criteria
  - Number of tuples in relation
  - Blocking factor
  - Number of distinct values
  - Selectivity of attribute (column)
  - ...

ProductID	Price
1	1000
2	1000
3	2000
4	1000
5	2000
6	1000

DBMSs decide their access strategies by statistics about the data stored. Their most important consideration is how many I/O operations are caused in the process. I/O means hard disk access – bringing data from storage into memory. This is the most time-consuming part of the entire query process.

Databases store table rows in blocks. Blocks are storage containers for database rows like crates are storage containers for apples.
DBMSs cannot bring a single row into memory without bringing the entire block it is stored in.

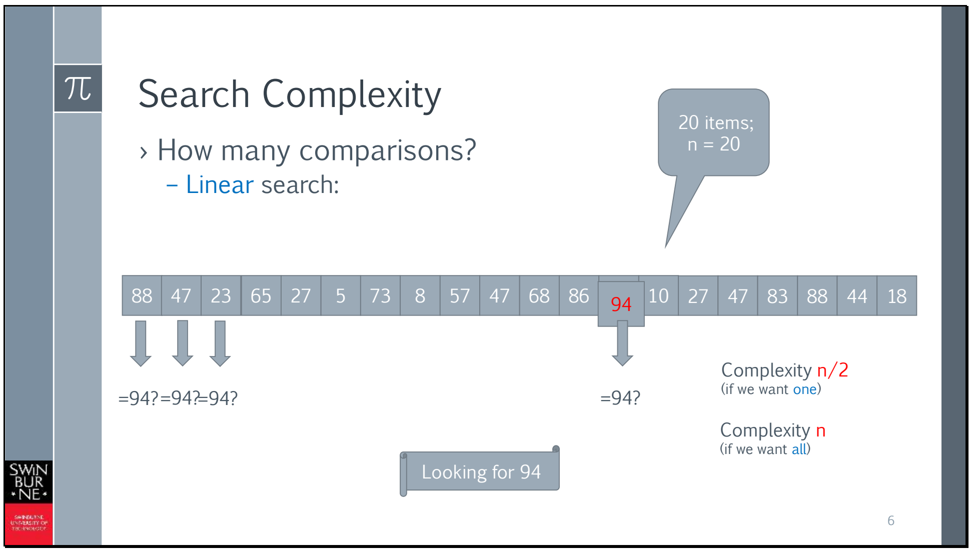
The blocking factor tells us how many rows are stored in a single block. If a table has 45 rows and the blocking factor is 15, the entire table can fit into three blocks. One I/O operation can usually bring 3 – 10 blocks, not just one, so it may be possible to bring an entire table into memory in one disk access.

The number of distinct values tells us how many different values an attribute has. In the example, the price attribute has only two distinct values (all the rest are duplicates). Selectivity is directly connected to this – the more different values an attribute has, the more selective it is. The DBMS assumes that if a column has two distinct values and we put a restriction on this column, 50% of the rows will match. This is clearly not true in the example, but the DBMS will assume it anyway.

If an attribute only discards 50% of the rows, it is not very selective. If an attribute is not selective, the DBMS will assume that most of the rows will be needed anyway, so it might as well bring the entire table into memory. If an attribute in a WHERE clause is very selective, the DBMS might decide to use an index instead, if it exists.

The blocking factor does not change over time, it has to be specified when the table is created. All other factors, however, will change over time. The DBMS will not update the statistics automatically. This would mean every INSERT, UPDATE or DELETE would trigger an update of the database metadata, which is clearly a performance issue.
Having outdated statistics, however, might lead to a wrong choice of access strategy. A decision has to be made when the Analyze command is run on a table.

Slide 6



Search Complexity

› How many comparisons?
  – Linear search:

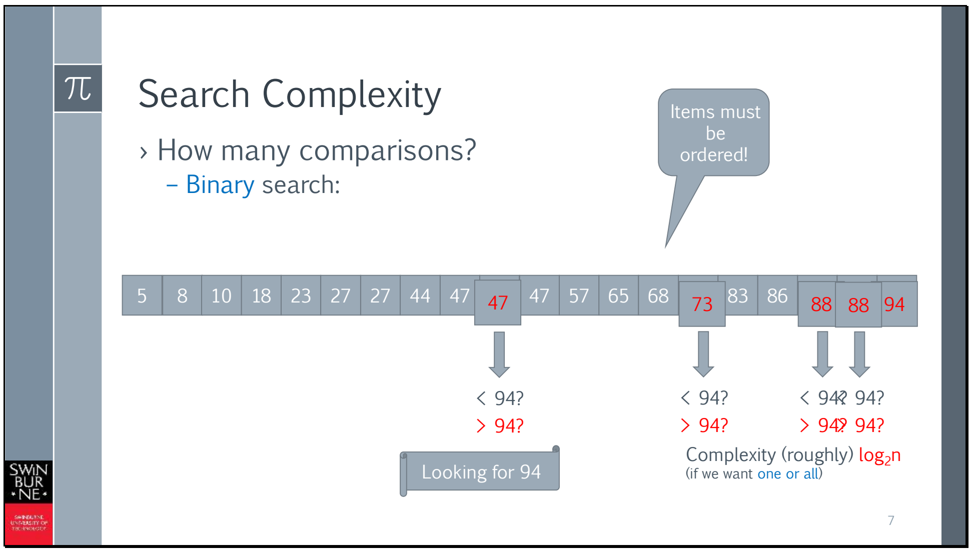
20 items; n = 20
88 47 23 65 27 5 73 8 57 47 68 86 94 10 27 47 83 88 44 18
=94?=94?=94?
=94?
Looking for 94
Complexity n/2
(if we want one)
Complexity n
(if we want all)
6

How much time does it take to find rows by restrictions in a table? This depends on how many comparisons we need to perform. The number of comparisons dictates the number of rows to bring into memory. Together with the blocking factor, knowing the number of comparisons helps us calculate the cost in I/Os of a query.

To calculate the complexity, suppose we have an unordered list – or an unordered database column – and we want to search this list for a particular value, in this case 94. Because the list is unordered, there is no option but to compare every value until we find the one we want.

This is very expensive – being a DBMS, we would have to bring all these values into memory. If there are millions of tuples in this relation, this can take a real long time. If we are looking for one value only, the average number of complexity is the number of entries divided by two, because some of the values will be found earlier in the list, some later, and on average, they will be found in the middle.
But suppose this is an age column in an employee table. If we want to find all our employees who are 94 (well maybe we want to send them birthday cards) we have no option but to compare all values with our desired value, and the complexity becomes n, the number of the entries.

Slide 7



Search Complexity

› How many comparisons?
 – Binary search:

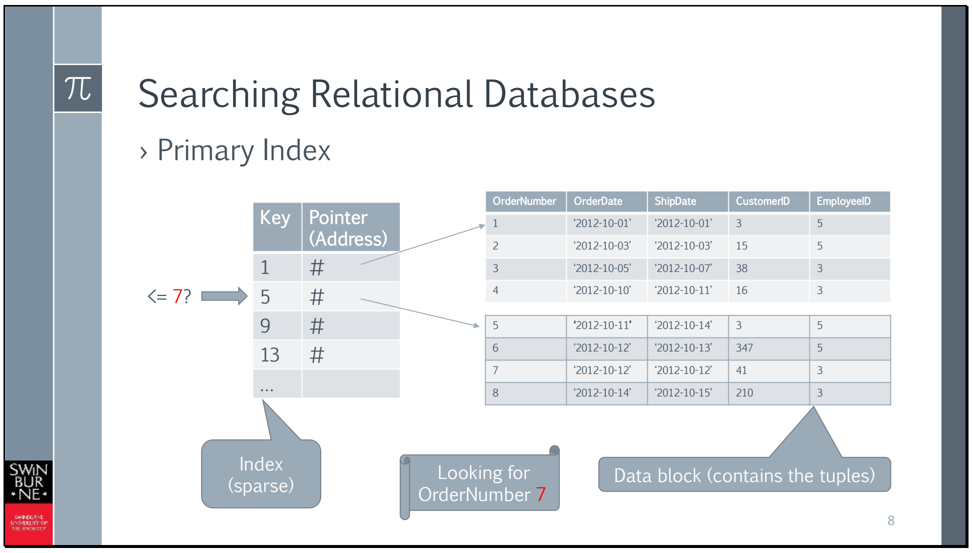
Items must be ordered!
5 8 10 18 23 27 27 44 47 47 47 57 65 68 73 83 86 88 88 94
< 94?
> 94?
< 94?
> 94?
< 94? 94?
> 94? 94?
Looking for 94

Complexity (roughly) $\log_2 n$
(if we want one or all)

7

Binary search is much faster, but it assumes that the items are in an ordered list. Binary search always looks at the middle element and then decides whether this item is smaller or larger than the value we are looking for. In this case, 94 is larger than 47, and because the items are ordered, we can discard the entire lower half of the list, including the value we compared.
We split the remaining values into halves again and compare with the middle value. The value we are looking for is still larger, so again, we discard the lower half of the remaining list. Since we keep dividing the number of remaining values to compare by two until there is only one left, the complexity is log2n.

Slide 8



Searching Relational Databases

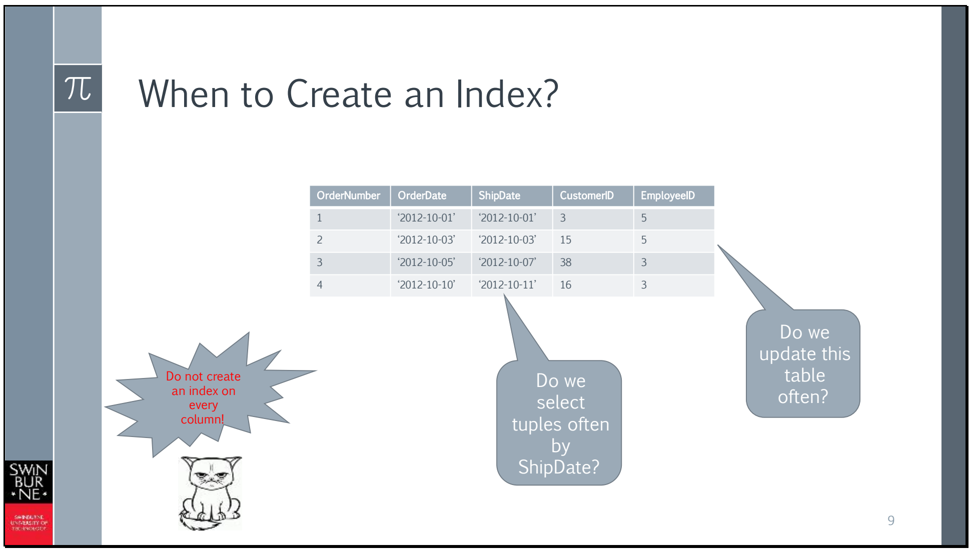
› Primary Index

Key	Pointer (Address)
1	#
5	#
9	#
13	#
…	

OrderNumber	OrderDate	ShipDate	CustomerID	EmployeeID
1	'2012-10-01'	'2012-10-01'	3	5
2	'2012-10-03'	'2012-10-03'	15	5
3	'2012-10-05'	'2012-10-07'	38	3
4	'2012-10-10'	'2012-10-11'	16	3
5	'2012-10-11'	'2012-10-14'	3	5
6	'2012-10-12'	'2012-10-13'	347	5
7	'2012-10-12'	'2012-10-12'	41	3
8	'2012-10-14'	'2012-10-15'	210	3

An index is a lookup table – like the index pages in a book – that lists the search value and a pointer (a memory address) to the actual data rows. The search value can be one or more attributes (columns) of the table the index is created on. Databases generally create a primary index – an index on a primary key - automatically when the primary key constraint is created. Many DBMSs store their data in primary key order. This way, the index can be sparse: It does not have to include a pointer to every individual tuple, only the row at the beginning of each block. The number of index entries then depends on the number of blocks needed to store the entire relation. This, again, depends on the number of tuples that fit into a block, which depends on the blocking factor.
Dense indexes have an index entry for every tuple in the table.
If we are looking for a value that is not mentioned in the index, such as 7, we can search the index using a fast access method – often even faster than binary search - and stop when we find a number that is smaller than or equal to 7. Since the DBMS has to bring the entire block into memory, it does not matter where in the block the row actually is.

Slide 9



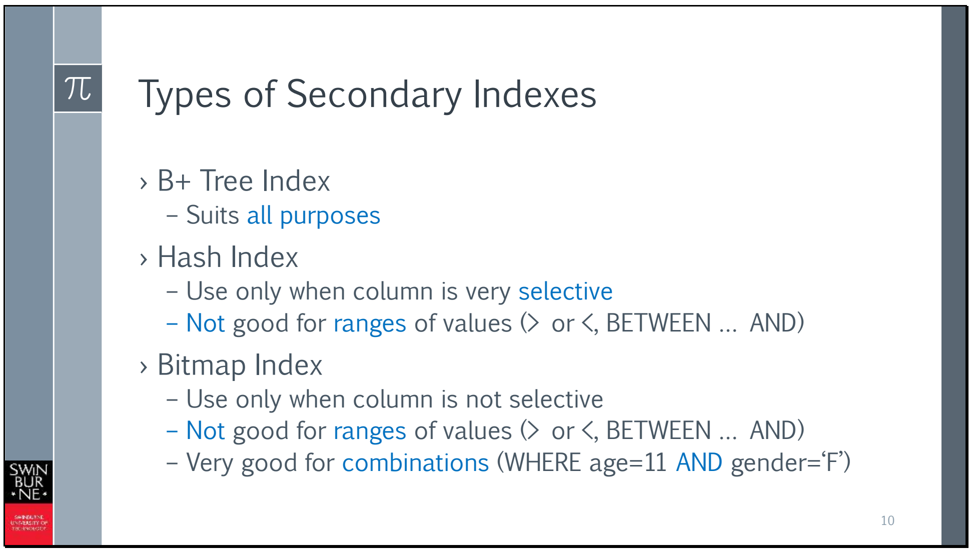
When to Create an Index?

OrderNumber	OrderDate	ShipDate	CustomerID	EmployeeID
1	'2012-10-01'	'2012-10-01'	3	5
2	'2012-10-03'	'2012-10-03'	15	5
3	'2012-10-05'	'2012-10-07'	38	3
4	'2012-10-10'	'2012-10-11'	16	3

Do not create an index on every column!
Do we select tuples often by ShipDate?
Do we update this table often?
9

Apart from the primary index, the user has to decide when to create secondary indexes. Indexes are created on attributes (columns). If you search by an attribute often, placing restrictions on it, an index may be useful. However, if you INSERT new rows often into this table, an index may do more harm than good.
Indexes have to keep their keys in sequential order, even when the table's tuples are not ordered by the indexed column. Whenever we add rows to a table, the index has to be updated to ensure the new key is in the correct place.
Also, if the attribute is not very selective, or if your queries return large numbers of rows, the DBMS may decide not to use the index. DBMSs offer tools for you to check how they execute a particular query, and it is recommended that you use these tools to see whether any new index is used for the query you think it would be useful for. If the DBMS chooses not to use it, you are better off dropping the index.

Slide 10



Types of Secondary Indexes

› B+ Tree Index
  – Suits all purposes

› Hash Index
  – Use only when column is very selective
  – Not good for ranges of values (> or <, BETWEEN … AND)

› Bitmap Index
  – Use only when column is not selective
  – Not good for ranges of values (> or <, BETWEEN … AND)
  – Very good for combinations (WHERE age=11 AND gender='F')

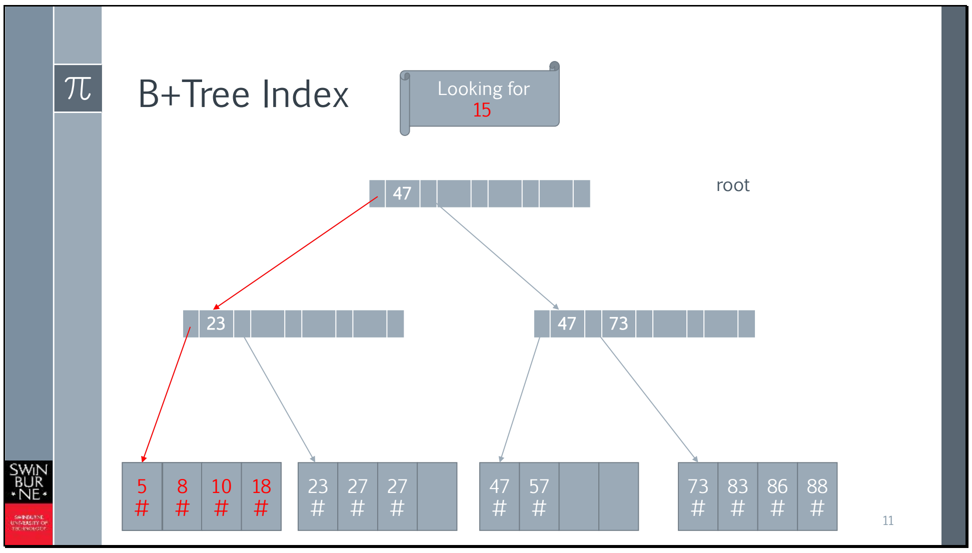
10

B+ Tree indexes are still the most popular indexing structure. Tree structures are quick to search, although updating them can take time. B+ trees are balanced trees that can deal with unique and non-unique values, selective columns and columns with few distinct values. B+ Trees will suit most columns, and they can search for ranges of values without problems.

Hash indexes are fast at retrieving single values. Hashing functions use the key of the index (the value of the attribute) to determine the storage address of an index value. Essentially, a mathematical operation, such as modulo, is applied to the attribute value, and the result provides the address of the index entry. Hashing is not very practical when many values are the same. The same key leads to the same address, and this is known as a clash. Because only one row can be at any address, clashes have to be resolved, which is an expensive operation. Hashing is a good option when we look for one or a few individual rows. Fetching a large range of values defeats the purpose, because each index entry has to be looked up individually.

BitMap Indexes are only useful if the columns indexed do not have many distinct values. A bitmap has to be created for each value (e.g. there would be separate bitmaps for age=11 and age=12). This is helpful in finding combinations of values quickly.

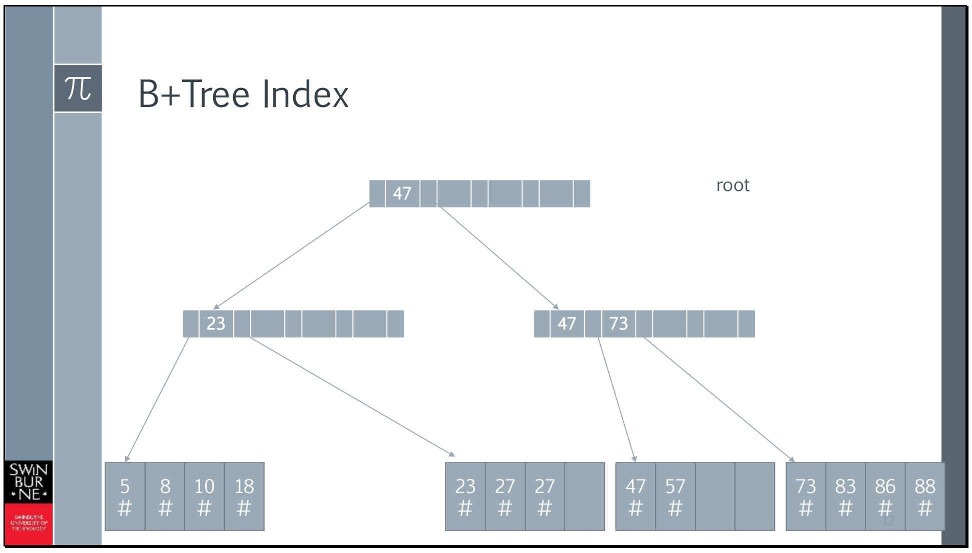
Slide 11



B+Tree indexes have leaf blocks (the lowest row in the picture) which contain values stored in order and their pointers to the data entries. The internal blocks (not leaf blocks) have pointers to lower level blocks. The pointers are between the values. A pointer to the left of a value points to a block that has smaller values than this value, a pointer to the right points to blocks with larger values. This makes for fast navigation of the tree.

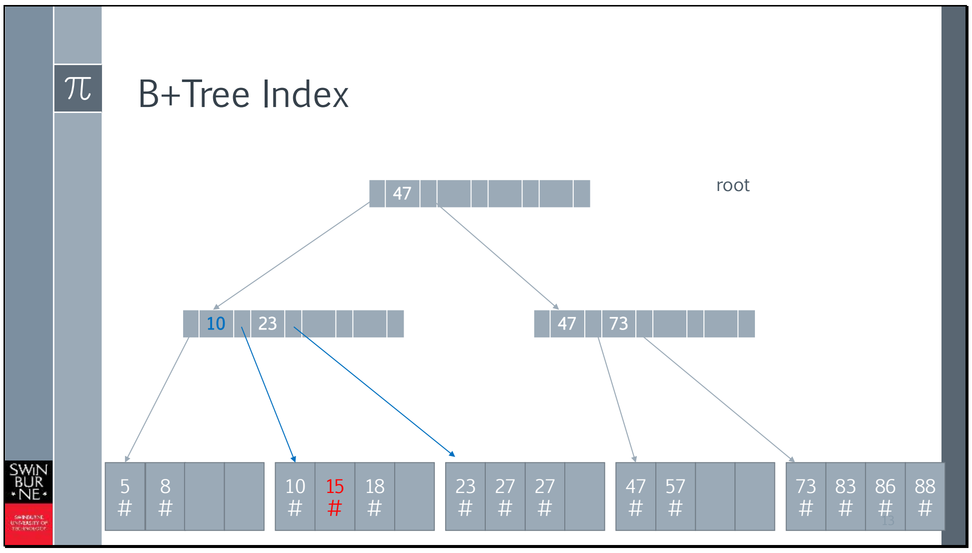
If we are looking for, say, 15, on the root level we know that 15 is smaller than 47, so we take the left block on the second level. 15 is also smaller than 23, so again, we take the left leaf block. There, we have to compare the numbers 5, 8, 10 and 18 to 15 before we can be sure it is not there.

Slide 12


B+Tree Index

In B+ Trees, the number of levels is not fixed, but the number of entries in each block is. The number of entries per block depends on the blocking factor.
If we have to insert a new value that belongs into a block that is full, we have to split the block.
As an example, we can insert 15. We can start by looking where it belongs in the leaf blocks. The entries in the leaf blocks have to be ordered by key, so we know exactly that 15 belongs in the left leaf block between the 10 and the 18. Because the block is full already, we have to split it into two. The split means that the next level of the tree is affected – we need another entry with a pointer to the extra leaf block. What do you think the result would look like? Pause the recording and think about it before you continue.

Slide 13



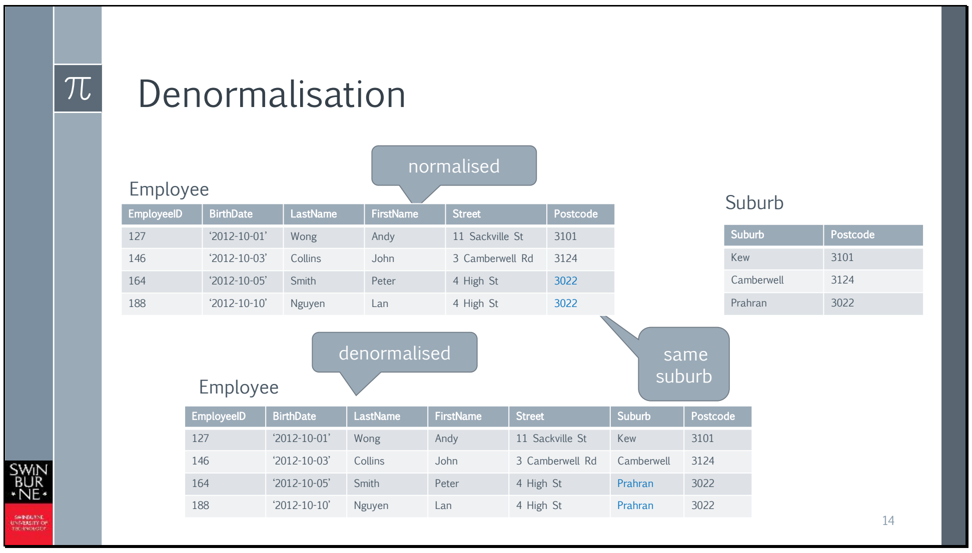
B+Tree Index

In B+Trees, when a block is split in half, half the entries remain in the old block and the other half goes into the new block.

So we have 5 and 8 in the leftmost block and 10 and 18 in the block beside it. 15 fits in between 10 and 18. Now we have to ensure there is a pointer to the new block from the next level up. Therefore we have to make an entry for the second block on the higher level. Since the second block starts with the key 10, we have to add 10 to the middle level block. Since 10 is smaller than 23, we first have to shift 23 one position to the right to make space for 10.

Note that the root level is not affected, because the block in the middle is not split.

Slide 14



Denormalisation

normalised
denormalised
same suburb

Employee

EmployeeID	BirthDate	LastName	FirstName	Street	Postcode
127	‘2012-10-01’	Wong	Andy	11 Sackville St	3101
146	‘2012-10-03’	Collins	John	3 Camberwell Rd	3124
164	‘2012-10-05’	Smith	Peter	4 High St	3022
188	‘2012-10-10’	Nguyen	Lan	4 High St	3022

Suburb

Suburb	Postcode
Kew	3101
Camberwell	3124
Prahran	3022

Employee

EmployeeID	BirthDate	LastName	FirstName	Street	Suburb	Postcode
127	‘2012-10-01’	Wong	Andy	11 Sackville St	Kew	3101
146	‘2012-10-03’	Collins	John	3 Camberwell Rd	Camberwell	3124
164	‘2012-10-05’	Smith	Peter	4 High St	Prahran	3022
188	‘2012-10-10’	Nguyen	Lan	4 High St	Prahran	3022

14

In relational database design, normalisation is important to avoid redundancy. Duplication can be avoided by creating a new table for the data that has duplicates and then linking this table to the initial table.

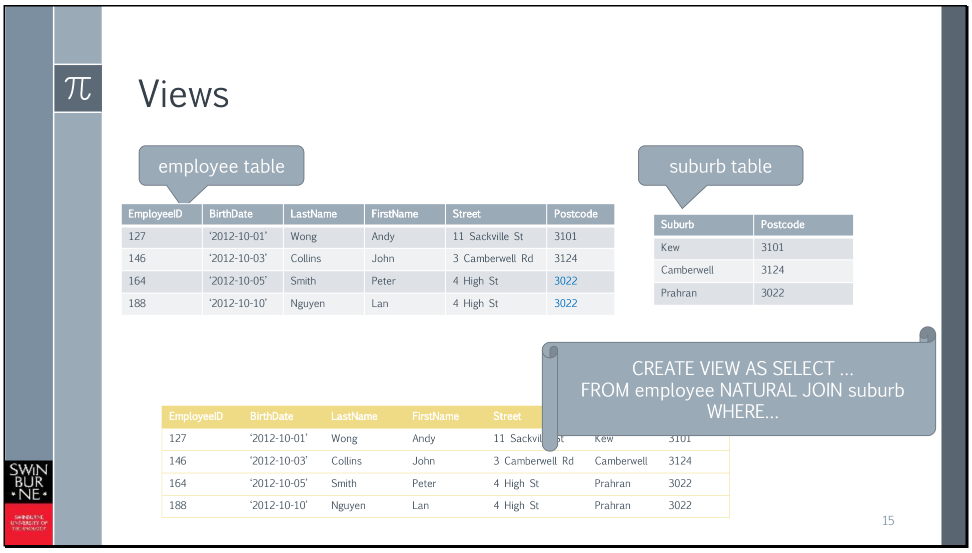
In the case of persons and their addresses it is often argued that addresses ought to form a separate relation, because several people might share the same address, and if we did not have a separate entry in another table, we might repeat the address.

Street addresses are rarely shared, but suburbs are shared often. According to the philosophy of relational modelling, postcodes and suburbs should be accommodated in different tables, because the relationship between the suburb and the employee is transitional – it depends on the street address of the employee. Having two tables means one of them has the postcode and the suburb, but the postcode still has to appear in the employee table, as a foreign key! This means that we would need a join operation if we want to query the database for employees and their addresses. The join is needed for just a single column (the suburb one). This does not make sense.

We should remember that creating many tables whose tuples are joined in every SELECT operation that accesses them is not good for performance. As we have seen, to join attributes from different tables, we have to do an extra search step that finds the matching rows. If a join happens often with this data, separating the tables is a bad idea. But if most SELECT statements only access the employees without the addresses, separate tables can be acceptable.

The case of suburbs is relatively straightforward – you may need to denormalise in situations that are not as clear cut and having two tables reduces duplication significantly.

Slide 15



Views

employee table

EmployeeID	BirthDate	LastName	FirstName	Street	Postcode
127	'2012-10-01'	Wong	Andy	11 Sackville St	3101
146	'2012-10-03'	Collins	John	3 Camberwell Rd	3124
164	'2012-10-05'	Smith	Peter	4 High St	3022
188	'2012-10-10'	Nguyen	Lan	4 High St	3022

suburb table

Suburb	Postcode
Kew	3101
Camberwell	3124
Prahran	3022

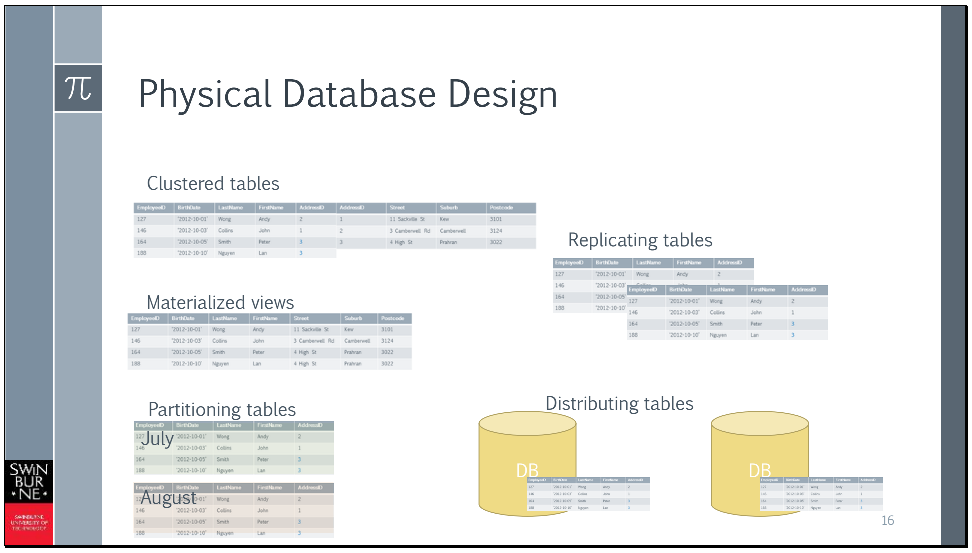
CREATE VIEW AS SELECT ...
FROM employee NATURAL JOIN suburb
WHERE...

EmployeeID	BirthDate	LastName	FirstName	Street		
127	'2012-10-01'	Wong	Andy	11 Sackvil st	Kew	3101
146	'2012-10-03'	Collins	John	3 Camberwell Rd	Camberwell	3124
164	'2012-10-05'	Smith	Peter	4 High St	Prahran	3022
188	'2012-10-10'	Nguyen	Lan	4 High St	Prahran	3022

15

If you decide that it is not meaningful to combine the tables, another way of speeding up joins of tables is to create views.
Views are SELECT statements that are executed whenever the view is accessed. Accessed means that the view is queried using a select statement, or even updated. When this happens, the DBMS first executes the original query that was stored using the CREATE VIEW statement.
The advantage is that the DBMS will create a query plan – a strategy how to carry out the query – when the view is first created. When the view is accessed, the SELECT statement that creates the view has already been optimised and can be executed directly, without the DBMS having to optimise it first.

Slide 16



Physical Database Design

Clustered tables

EmployeeID	BirthDate	LastName	FirstName	AddressID	AddressID	Street	Suburb	Postcode
127	'2012-10-01'	Wong	Andy	2	1	11 Sackville St	Kew	3101
146	'2012-10-03'	Collins	John	1	2	3 Camberwell Rd	Camberwell	3124
164	'2012-10-05'	Smith	Peter	3	3	4 High St	Prahran	3022
188	'2012-10-10'	Nguyen	Lan	3				

Replicating tables

EmployeeID	BirthDate	LastName	FirstName	AddressID
127	'2012-10-01'	Wong	Andy	2
146	'2012-10-03'	Collins	John	1
164	'2012-10-05'			
188	'2012-10-10'			

EmployeeID	BirthDate	LastName	FirstName	AddressID
127	'2012-10-01'	Wong	Andy	2
146	'2012-10-03'	Collins	John	1
164	'2012-10-05'	Smith	Peter	3
188	'2012-10-10'	Nguyen	Lan	3

Materialized views

EmployeeID	BirthDate	LastName	FirstName	Street	Suburb	Postcode
127	'2012-10-01'	Wong	Andy	11 Sackville St	Kew	3101
146	'2012-10-03'	Collins	John	3 Camberwell Rd	Camberwell	3124
164	'2012-10-05'	Smith	Peter	4 High St	Prahran	3022
188	'2012-10-10'	Nguyen	Lan	4 High St	Prahran	3022

Distributing tables

Partitioning tables

July

EmployeeID	BirthDate	LastName	FirstName	AddressID
127	'2012-10-01'	Wong	Andy	2
146	'2012-10-03'	Collins	John	1
164	'2012-10-05'	Smith	Peter	3
188	'2012-10-10'	Nguyen	Lan	3

August

EmployeeID	BirthDate	LastName	FirstName	AddressID
12	[illegible]-01'	Wong	Andy	2
146	'2012-10-03'	Collins	John	1
164	'2012-10-05'	Smith	Peter	3
188	'2012-10-10'	Nguyen	Lan	3

DB

EmployeeID	BirthDate	LastName	FirstName	AddressID
127	'2012-10-01'	Wong	Andy	2
146	'2012-10-03'	Collins	John	1
164	'2012-10-05'	Smith	Peter	3
188	'2012-10-10'	Nguyen	Lan	3

DB

EmployeeID	BirthDate	LastName	FirstName	AddressID
127	'2012-10-01'	Wong	Andy	2
146	'2012-10-03'	Collins	John	1
164	'2012-10-05'	Smith	Peter	3
188	'2012-10-10'	Nguyen	Lan	3

Most DBMSs offer a large variety of ways of physically arranging the data in permanent storage – on disk - so that it is fast to retrieve.

Clustering related tables that are often joined – such as the employee table and its addresses - is one way of pre-matching tuples so that the DBMS does not have to search for matches when a JOIN is required. It means that rows from one table and the related rows from another table are stored in the same block. This ensures that related rows from different tables are always fetched at the same time.

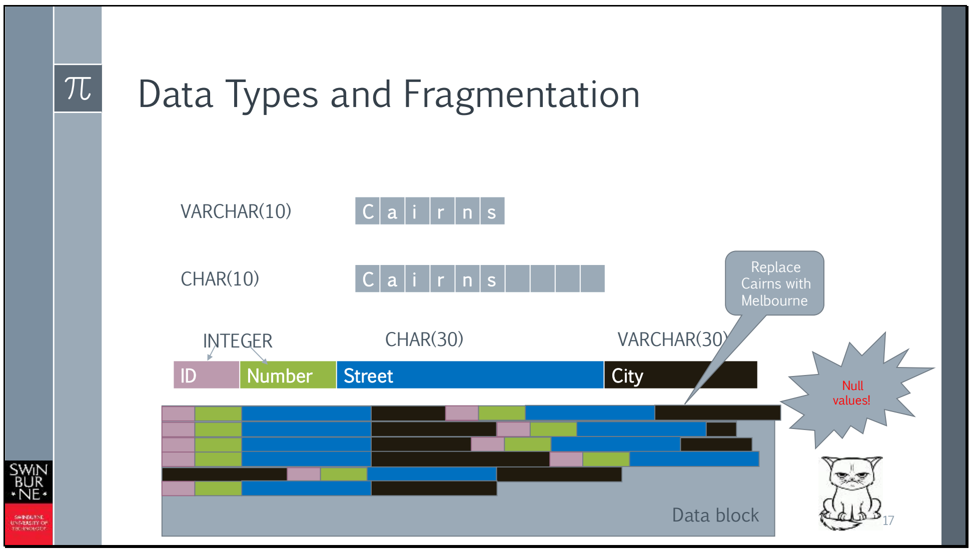
Materialized views are physical copies of the original data. They can be created like normal views – using a SELECT statement. This is a way of copying related data from different tables into the same blocks. Making a copy also means that the original data does not have to be accessed when a query is issued.

Copying the data means replication. Replicas are an opportunity to distribute the query load. The problem with replication is how do we keep both copies identical. The more often we update replicas, the more work the DBMS has to do, and the busier it becomes. Queries have to wait. If we do not update often, the replica may have stale data.

Replication is often used in distributed approaches, where copies of the database are located on different hardware. When more processors are available, the work load is split up and can be executed in parallel.

All of these approaches have advantages and disadvantages. They require careful planning and consideration of the queries and updates involved. Physical database design is very relevant for performance, but it is a vast topic that cannot be covered in detail in this unit. It is mentioned here so you are aware of these options when the necessity arises.

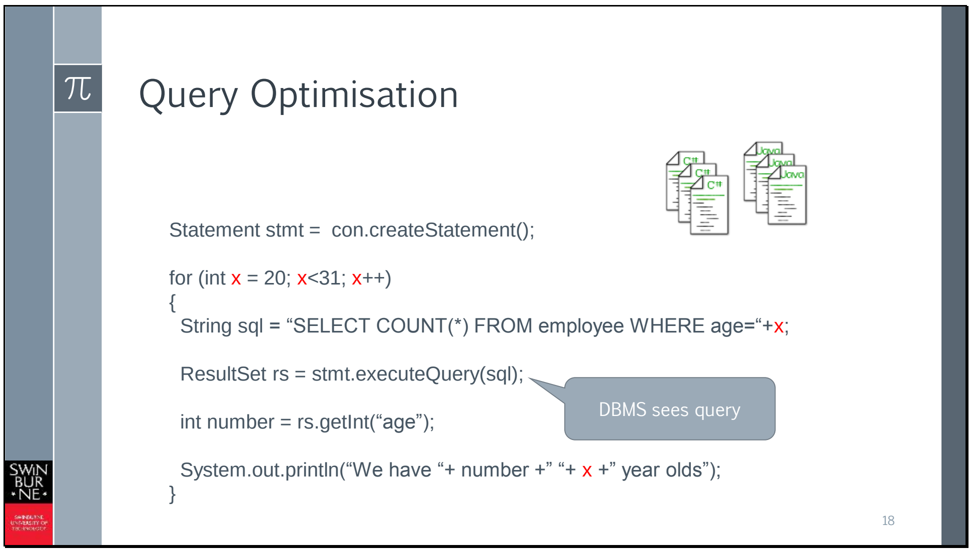
Slide 17



Data Types and Fragmentation

Different DBMSs offer different data types. Probably all of them offer CHAR and VARCHAR. CHAR is fixed length, VARCHAR is variable length. A CHAR of 10 occupies 10 bytes in memory, VARCHAR 10 occupies as many bytes as letters are stored, but no more than 10.
When new rows are inserted, the database usually leaves a buffer between rows, but if the VARCHAR field is later updated, the buffer may be exceeded and the row has to be moved. This leads to fragmentation, and more disk access is needed to fetch consecutive rows. The same applies with other data types of variable size.
The effect of this problem is even worse when you put null values into a variable length field initially. When you insert a new row and the city field is null, the length is zero. When you later fill in the value, the row may no longer fit into the allocated space and might have to be moved. If you have the option, fill in all attribute values with variable size in the INSERT statement rather than updating the variable length fields later.

Slide 18



Query Optimisation

```
Statement stmt =  con.createStatement();

for (int x = 20; x<31; x++)
{
  String sql = “SELECT COUNT(*) FROM employee WHERE age=“+x;

  ResultSet rs = stmt.executeQuery(sql);

  int number = rs.getInt(“age”);

  System.out.println(“We have “+ number +” “+ x +” year olds”);
}
```

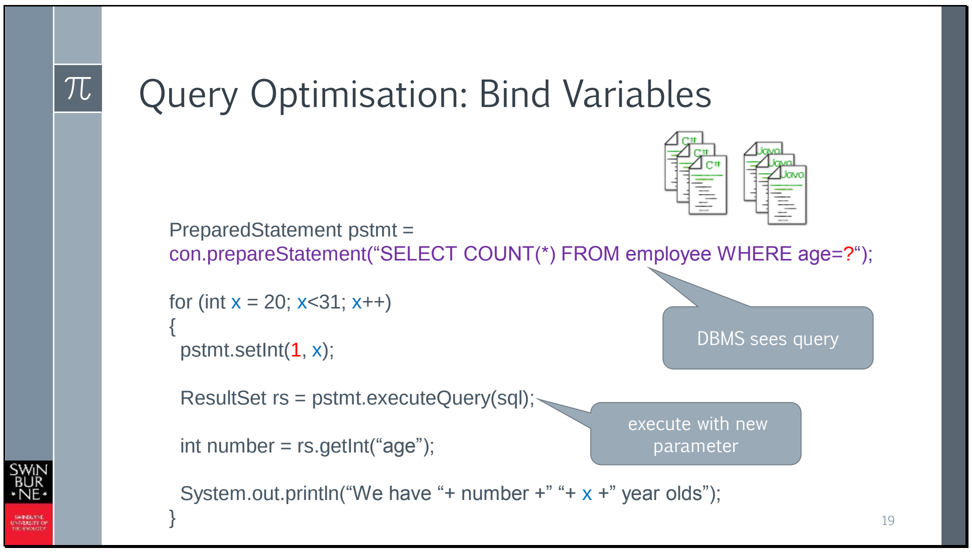
DBMS sees query
C#
Java

Embedded SQL is SQL used from within a program. Every programming language has libraries for database access. Any useful database access library gives you two options to implement queries. It is important that you understand the difference.

In this example, we are querying the database for the number of 20-year-old, 21-year-old, etc. employees. For every age, we create a new SQL string which contains the newly incremented age.   This query is sent to the database when we run the method executeQuery.

What is the problem with this approach? Every time the DBMS receives the query, it looks different: the number in the restriction has changed. In each loop, the DBMS considers the query with the new age as a separate query and submits it to the query optimiser. As we know, the query optimiser returns a strategy for the table access, even when the query is simple. Because the query structure is the same, the access strategy will also be the same every time. How can we stop the DBMS from re-optimising the same query over and over?

Slide 19



Query Optimisation: Bind Variables

```
PreparedStatement pstmt =
con.prepareStatement(“SELECT COUNT(*) FROM employee WHERE age=?“);

for (int x = 20; x<31; x++)
{
  pstmt.setInt(1, x);

  ResultSet rs = pstmt.executeQuery(sql);

  int number = rs.getInt(“age”);

  System.out.println(“We have “+ number +” “+ x +” year olds”);
}
```

DBMS sees query
execute with new parameter

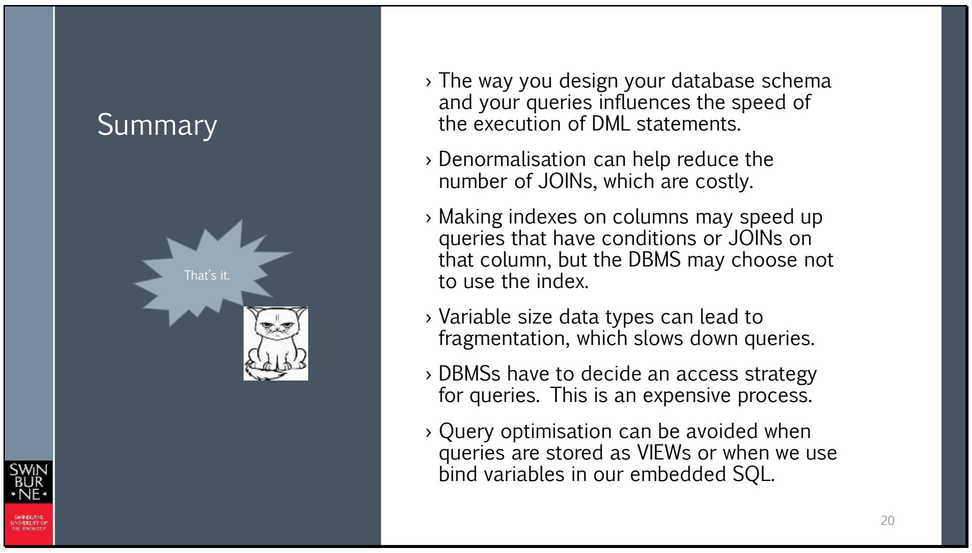
The answer has many names; some authors and database products talk about bind variables or prepared statements, and some say we are using static instead of dynamic queries.

The important difference is that in this static query that uses bind variables, we send the query to the DBMS for optimisation first. At this time, we do not state a literal value but a variable in the WHERE clause. In Java, the variable is represented by a question mark. In C#, you would be using a variable name preceded by an @ sign.

Later, when we run the query to retrieve the results in a loop, we substitute the variable with a literal value. We set the value on the prepared statement. As there can be more than one variable in an SQL string, we indicate the how manieth question mark we are replacing with the value x. In this case, we only have one place holder question mark, so the number is 1. (Yes, the counting starts at 1 not zero in this case.)

When you use this approach, your query is optimised once and executed many times without re-optimising. This speeds up the query.

Slide 20



Summary

- The way you design your database schema and your queries influences the speed of the execution of DML statements.
- Denormalisation can help reduce the number of JOINs, which are costly.
- Making indexes on columns may speed up queries that have conditions or JOINs on that column, but the DBMS may choose not to use the index.
- Variable size data types can lead to fragmentation, which slows down queries.
- DBMSs have to decide an access strategy for queries. This is an expensive process.
- Query optimisation can be avoided when queries are stored as VIEWs or when we use bind variables in our embedded SQL.

20

Here are the most important points discussed in this module. You may want to stop the recording to have a read through them. When you are ready, start the quiz about this module.