Contents

1 — Database Systems

Databases and Database Systems

Databases and database systems are everywhere. It is safe to say that it would not be possible for you to read this pdf (or attend the lecture in which it is used) without the involvement of databases. They store data in an orderly fashion and allow for its retrieval at a later date. Thus a database keeps track of what students are attending the university now, who is enrolled in what course, what PDF files are associated with what term's offering of the course, and so on. The more traditional examples relate to everyday life for a common person: banking, airline reservations, e-commerce websites...

A database is by no means the only way to store data; data may be stored in any format one wishes. Professors, for example, are notorious for storing their data in large unorganized piles of paper in their offices. Simple text files are often used to record data. A surprising amount of the world operates on the basis of Microsoft Excel spreadsheets. This is not because of the mathematical functions or intelligence of Excel but rather the gridlines. Seriously.

For small amounts of data it may not matter very much that the data is not structured and not organized. For larger amounts of data, this quickly becomes impractical. Some sort of structure is needed and the first approach to dealing with this was to use files. Programs stored data in files and manipulated those files. Suppose a simple example from [SKS11], about keeping track of students at university.

Assuming there are files for students, instructors, courses, et cetera, then we need programs (or functionality of a single program) to add students, add instructors, add courses, register students for courses, generate class rosters, assign grades to students, generate transcripts... As time goes on, new functionality is added and the functionality is extended and changed.

Files for data storage have some serious drawbacks:

- **Data Redundancy:** The same data may be duplicated (appear in more than one location in the files).
- **Data Inconsistency:** Copies of duplicate data may not agree (e.g., an address update for a person may not be reflected in all copies).
- **Data Isolation:** Data can be spread over lots of files and hard to get together to find the information you want.
- **Integrity Problems:** There are real world rules that need to be respected, such as the pieces of inventory in stock cannot become negative; files don't support rules.
- **Atomicity Problems:** Updates to data should be atomic: succeed completely or it should be as if it did not happen at all. File systems may, or may not provide guarantees about writing to them.
- **Concurrent-Access Problems:** If two operations take place on a file at the same time, they may, or may not produce the expected outcome (remember earlier courses about concurrency?)
- **Security Problems:** It is hard to restrict access to just a subset of data: a professor should be able to only update the grades of students taking her class in the current term, for example...

The database, or more specifically, a database management system (DBMS) is a solution to these problems. Of course, a database, if poorly designed, will not fix all of those problems. It is certainly possible to have redundant or inconsistent data if the data storage is poorly designed. It is similarly possible to fail to introduce the rules to ensure integrity and suffer as the logical rules are violated.

There are three levels of abstraction we are interested in when discussing a database, from least abstraction to most [SKS11]:

1. Physical Level – How the data is actually stored.
2. Logical Level – How the data is structured (organized).
3. View Level – A virtual structure based on how we would like to see the data.

When examining the physical level, it matters how the data is stored (usually on disk), including the low level data structures to contain the data. Database administrators may care about this data and it may be necessary to manipulate the physical storage configuration to get better performance, But for the most part, application programmers are unaware of this level; they focus on the next level.

The logical level tells us what data is in the database and what the relationships are between those pieces of data. Because of the abstraction, when thinking about the database, there is no need to be aware of the physical level data structures or even their location. At this level data that belongs together is grouped together, in the same way that we might pack data together in a structure or class.

At the view level, we are looking at only a part of the database. There are often many views for the same database, such as one for students and another for instructors and yet a third for university administrators. We will create lots of different views of the same data based on the needs of the users (or database administrators who want to check on things).

A database is typically accessed via a database server. Thus we see the client-server model in use: an application program then accesses the database through some request interface and receives answers. There also exist client programs that allow you to send commands to the database directly and get responses. This is one way that we might see what the current state of the database is. We can also issue statements that change the data directly, although caution must obviously be used to ensure correctness; an incorrect statement can do a lot of damage and lead, perhaps, to unrecoverable data loss. Soon enough, we will begin to learn the format and syntax for issuing database commands.

Schemas & Instances

A database *schema* defines the structure of the database, including what data is to be stored in what format. An instance of a database reflects both the schema and the particular content of that database. If this analogy helps, you may think of a schema as being like a class file definition in an object-oriented programming language. Except the schema can define multiple types of object and the database instance includes instances of many different types.

A (logical) schema may look something like the example below, which represents some motor vehicle registration data. This is the sort of thing that ministries of transport keep on hand so they can identify vehicles and their owners and their license plates. Keep in mind that the design shown is a choice; there are other options for how it could be organized.

VEHICLE

VIN	`string(17) not null, primary key`
year	`integer not null`
make	`string(64) not null`
model	`string(64) not null`
license_plate_number	`string(8)`

LICENSE_PLATE

number	`string(8) not null, primary key`
expiry	`date`
owner_address_id	`integer`

OWNER_ADDRESS

id	`integer not null, primary key`
name	`string(64) not null`
street	`string(64) not null`
city	`string(32) not null`
province	`string(2) not null`
postal_code	`string(7) not null`

There are some immediate observations. The first is these three structures (vehicle, license plate, and owner address) are all intended to represent some analogue to a real life thing in the same way that objects in an object-oriented programming language do (or structures in a language like C). Each entity is broken down into some number of fields, and each field has a name and a type. Types have some meaning, so a year must be an integer, like "2016", and cannot be a string or contain a decimal. There are potentially restrictions on a type, such as the "not null" requirement. Field names must be relatively unique but do not have to be globally unique. There also appear to be data elements that share some names with other data elements or other entities. That is a subject we will return to later.

There are also some items of meta-data (data about data) in the shown schema. For example, `id` is shown as "primary key", which provides additional information and additional restrictions. There are also indexes[1] that are used in searches. They can be added, removed, and modified, without affecting the underlying data.

No information is shown here about how the physical schema looks; how is this data stored in a practical sense? From the logical view it is irrelevant. If we wish, we can move some frequently accessed data to a faster device (SSD?) without affecting the logical schema and therefore without affecting any program that relies on this data.

If we create a new database and apply this schema, we created a new instance but it is empty. This is the initial state of the database and each modification to the database creates a new state of the database. At any time, we can take a snapshot of the current database (export all its data). This is a common operation for taking backups, for example, of the data, or just to make a copy to use on another test system.

The most common type of modification would be to manipulate data in it: add an item (e.g., a new car is purchased and the data gets entered), modify an item (e.g., someone moves and their address needs to get updated), or delete an item (e.g., a license plate is retired from service). This kind of change does not modify the schema, but it does modify the data. If we attempt to modify the data in a way that is not consistent with the schema (e.g., try to put "ABC" in an integer field), that modification will be rejected. Nevertheless, care must be taken in what data updates are issued, otherwise we may overwrite or delete important data. An incorrectly-crafted update command can easily cause unintended damage such as setting all VINs to be the same value. Careful schema design, as we will see, will prevent that sort of error, such as introducing a requirement that VINs be unique. If that rule exists, the update is rejected. Design cannot prevent all such errors, unfortunately. There can be two people with the name "Thomas Anderson"[2], so no rule of unique names can be introduced, and an erroneous update command could set all names to exactly that even if the schema is designed perfectly.

Data modification is not the only kind of change, however, because the schema can and does change. Textbook authors like in [EN11] sometimes tell you that schema changes will be rare, but much depends on your application. If it is a stable application then schema changes are rare; if it is a startup company or an app with ever-expanding functionality then the the schema will change regularly. Schema changes are design changes, though, and they do not typically happen in the normal operation, but instead happen during version upgrades.

Modifications to the schema can be made in a way that does not affect the data or require changes to applications that access it. If a new field "country" is added to the address entity, then something that looks only for the name will not be affected. Similarly, a new entity, unrelated to these, such as "Service Ontario locations" (places where drivers need to wait in agonizing lines to register vehicles, for example) can be created with no impact on the existing data.

[1] Actually, indices is the correct plural, but "indexes" is used very commonly in English

[2] Whoah.

Modifying a schema, can, of course, be dangerous. If we decide to remove a data element, such as the province from the address, then data may be lost. If we add another property, such as a country, by default it contains nothing – and if this property is to be defined as "not null" then a default value is needed.

The database server can, of course, have multiple instances, and multiple schemas[3] all of which are independent of one another. Two databases may use the same schema but contain completely different data. Two databases can begin with the same schema but evolve in different directions over time based on the changes that are made. A change in the schema may result in changes to some data (e.g., data is truncated when a field is made smaller) but does not have to (e.g., a field is made larger and the existing data is unchanged).

The entities (vehicle, license plate, owner address) themselves are only a part of the picture. There are also the relationships between these, which are partially reflected in the fields like `owner_address_id`. This takes us to our next subject, which is to examine the relational model in more detail.

[3] The plural of schema is schemata, but I think we know how this goes by now.

2 — Entity-Relational Model

Database Data Models

There is a certain level of abstraction provided by the database; that is, the users, even application programmers, may not need to be aware of the way in which the data is stored or handled. With that in mind, there will be a *data model* – the abstraction that describes the structure of the database [EN11]. The data model is not only how the database is structured in terms of its schema, but also defines some basic operations to access and modify data (and the schema).

According to [SKS11] there are four rough categories for data models. For the most part we will focus on the first one, the Relational Model. We will also give some consideration to the Entity-Relational model. The other two, Object-Based and Semistructured, will be noted for completeness but are not the focus of much discussion.

Relational Model. The relational model uses tables to represent data (and there are also tables that contain the relations between those tables). Tables have unique names, and are composed of columns (and column names must be unique per table). An entry in such a table is a row. This is a record-based model in that each table comprises fixed format records of specific types. A record definition says what fields exist and their types (and we have seen that already). Each column in the table is one of the fields [SKS11].

Entity-Relationship Model. The entity-relationship model (sometimes abbreviated E-R or ER) model takes the idea of structured types, like a `struct` in C. Entities have various properties and there are also relationships. Entities are like objects in an object-oriented programming language in that they are supposed to correspond to a cohesive "thing" with properties [SKS11].

Object-Based Data Model. This model borrows from Object-Oriented Programming to expand it to include the ideas of encapsulation, methods, and object identity, to build on the entity-relationship model [SKS11].

Semistructured Data Model. The semistructured model is somewhat more flexible than various other models because the attributes may vary. XML is a good example of a semistructured data model. In XML we can easily have a type where things are arbitrary length and types of attributes may vary.

Entities and Relationships

We have already discussed the idea of entities as objects. A vehicle has a VIN, a year, a model, a colour, et cetera. Every vehicle has those attributes, and some of them will be unique (e.g., VIN) to one vehicle and others will be shared (e.g., Volkswagen makes something like 1 000 000 Golf model vehicles in a year).

In addition to this, we have the concept of a relationship between identities: a vehicle is owned by a person. This is a simple association. The definition of our associations can also contain some important "rules" about our data. For example, a vehicle may have only one owner.

To speak more formally about entity relationships we will use some mathematical notation (mostly set notation). The relational model traces its way back to a paper published in 1970 that described the database in mathematical relationships and first-order logic [Cod70]. The math may seem intimidating when we first examine it, but

ultimately provides an unambiguous and concise way of describing how it all works, even if you don't necessarily adopt that as a way of thinking about the problem. The mathematical description is embodied in a table.

We need to spend some more time to define some terminology, specifically: domain, tuple, attribute, and relation.

A *domain* D is a set of atomic values, where atomic means that the value is not divisible in the relational model [EN11]. A telephone number like (212) 867-5309 may be divisible in the sense that we can identify 212 as the area code, but as far as the database is concerned if a phone number is ten digits it will consider that ten digit phone number to be indivisible. If the database designer intends for area codes to be separate from the remaining digits of the phone number that is a design choice and there will accordingly be two separate fields for it.

As we already saw in the previous examples about vehicles that fields, domains, have a data type associated with them such as integer, string, et cetera. Oftentimes there is a specified length of the field, such as limiting a phone number to 10 characters.

To indicate that a value is missing, unknown, or not relevant, there is the possibility for a domain to be `null`. This is a member of every domain by default, although it is possible to specify that a null is not permitted in the format. The use of null can occasionally cause us some headaches in the same way that null being used as a sentinel value in Java, for example, can lead us to a `NullPointerException`. One example of the way that null causes problems is that null does not equal null...

In some terminology that might be slightly confusing, a row in a table represents a *relationship* between a set of values: that is to say that things like "Volkswagen" and "Golf" and "2015" all go together. This is what we call a *tuple*, which corresponds mathematically with an *n-tuple* (that is a tuple with n values in it) [SKS11].

The table itself we call a *relation* (which is why this is the "relational" model). An *attribute* is then a column in that table. So if the relation is an address, the attributes will be name, street, city, province, postal code...

Let's recap this mathematically, as in [EN11]: A relation schema R is denoted as $R(A_1, A_2, ..., A_n)$. It is named R and has a list of attributes A_1 through A_n. Each attribute is the name of a domain D. The degree of the relation is the number of attributes. This is the table definition, just explained in a mathematical way.

The table content, or the relation itself, is denoted $r(R)$ and it is the set of n-tuples where $r = \{t_1, t_2, ..., t_m\}$. Each n-tuple is an ordered list of values where each value v_i is an element of the domain of attribute A_i or the null value. We might reference these as $t[A_i]$, $t.A_i$, or $t[i]$.

So if our relation schema R is as follows:

OWNER_ADDRESS

id	`integer not null, primary key`
name	`string(64) not null`
street	`string(64) not null`
city	`string(32) not null`
province	`string(2) not null`
postal_code	`string(7) not null`

Then the relation might look something like this:

OWNER_ADDRESS

id	name	street	city	province	postal_code
24601	Jean Valjean	19 Rue des Prisonniers	Ottawa	ON	B1B 1B1
25981	Thomas Anderson	1234 Main St	Waterloo	ON	A0A 0A0
12949	Alice Jones	4 Generic Place	Kenora	ON	C2C 2C2

There are 3 tuples corresponding to three addresses. You may notice that they are in no particular order. This is normal. A relation is a mathematical set, which unlike a list, has no intrinsic ordering. They will be stored in some order in physical storage, but no guarantees are made about that. We can ask for elements to be sorted, if we want to request data from the database, but because it is a set, order is not relevant.

A relationship is an association among several entities: license and address have a relationship and that relationship we can call "ownership". This is simple enough to understand.

A relationship set is a mathematical relationship set is a mathematical relation among $n \geq 2$ entities:

$$\{(e_1, e_2, ..., e_n) | e_1 \in E_1, e_2 \in E_2, ..., e_n \in E_n\}$$

where $(e_1, e_2, ..., e_n)$ is a relationship [SKS11].

We might have a rather unwieldy relationship: between vehicle and license there are a lot of columns. A license plate is uniquely identified by its numbers and/or letters and a vehicle is uniquely identified by its VIN. To prevent redundant (duplicate) data, we prefer that we use just their unique identifiers. So if the vehicle with the VIN 5N1BA0ND5BNF18322 currently is associated with the plate ZZZZ999, the relation will be (ZZZZ999, 5N1BA0ND5BNF18322) (or the other order is fine, it has the same meaning).

In the relational model the relationship representing ownership will itself be a relation (a table). It will contain two columns: one for the license plate number and one for the VIN. A relationship may possess attributes of its own, that is, ones that do not directly reference the attributes of other relations (tables). As an example, the relationship may have a third attribute that contains a date and time when the relationship was last modified (e.g., the vehicle is sold and it is associated with a new plate).

The most common of relationship is binary: it involves two entity sets; a vehicle has an owner and there is a mapping between the two. A particular table may be involved in arbitrarily many relationships. In the simple vehicle example, an address is associated with a license plate. A license plate is in turn associated with a vehicle. We are not restricted to binary relationships; a relationship between multiple entities can be created. For example, if we were to track insurance policies in our database, given that vehicle insurance is mandatory, we could have yet another table that has the attributes of (id, VIN, insurance policy number, address id), referencing both the vehicle and the address tables.

Keep in mind that sometimes a mapping will not (yet) exist. A vehicle may be entered into the database before it has been assigned a license plate, for example.

Constraints

There are rules, there are always rules. In the relational model we call restrictions or rules constraints. These constraints are supposed to reflect the logical state of the world. We could divide these constraints up into one of three categories [EN11]:

1. Constraints inherent in the data model.
2. Explicitly added constraints.
3. Constraints that cannot be captured in the data model.

Some of the constraints that are inherent in the data model are relatively easy to explain. If we have defined the license plate field as a maximum of 8 characters, this adds an implicit constraint that means any attempt to assign a license plate that is too long (say, 10 characters), will be rejected. Thus this constraint is enforced just by the definition of the data storage method.

Other constraints must be explicitly added. We can specify that a column is "not null", or that the content must be unique. In both of these cases, again, an attempt to set an invalid value will result in the change being rejected. Similarly, we may impose rules on relationships: we can say that a license plate must be associated with an address: the field for the owner address ID must match to a row in the owner address table.

Some constraints, unfortunately, cannot be described in the data model. We may specify that a postal code is six characters in length, and we may say it cannot be null (empty) but there is no good way to specify the format

must follow the Canadian postal code format (e.g., A1B2C3). These constraints, if they are to be enforced, must be enforced outside of the database. Such things are sometimes called business rules.

Cardinality constraints express relationships in a structured way. Our choices are:

- One to one (1:1).
- One to many (1:N).
- Many to one (N:1).
- Many to many (N:M).

It should be easily possible to think up real-world examples for each of the cardinalities. One license plate maps to one vehicle and vice-versa. One imagines by this point there is no need to put together drawings and fancy explanations of the mappings.

Keys

We need a way to identify tuples uniquely in terms of their attributes. That is to say, no two tuples in the table are permitted to be exactly the same in every column. A *super key* is a set of one or more attributes that uniquely identify one tuple in the relation (table) [SKS11]. By default, of course, the full set of attributes will be one such super key. Some attributes on their own will be a super key, such as VIN, because VINs are unique. Others are not: names are not unique. But a super key may be something like (VIN, make, model). We can take a super key and add to it and it will still be a super key. But what we would like to do is take away some things instead.

If a super key is as small as it can be but no smaller, then it is a minimal super key and we call it a *candidate key*. A candidate key, more formally, has no proper subset that is a super key [SKS11]. So, (VIN, make, model) is a super key but not a candidate key; VIN on its own is a candidate key.

There are then, potentially, several candidate keys. A student at the University of Waterloo can be identified uniquely by both a student number (e.g., 20000000) and a userid (j9999doe). Both of those are candidate keys for identifying an individual student. The database designer will choose a *primary key* from the candidates to be the main way of uniquely identifying a tuple.

Soon enough we will spend some time to discuss good database design. Typically, however, the primary key will be one field, to reduce duplication and confusion. We've seen a little bit of that already in that we assigned addresses an "ID" field (of type integer) which is not a part of the actual address; it won't get printed on documents or even necessarily be user visible, but it is handy for referring to an address uniquely in a compact way. Our primary key should be something that is always present and always unique.

When we have a join relation the primary key may be formed by the two entities being referenced. If the relation has a tuple (ZZZZ999, 5N1BA0ND5BNF18322) then that itself can be the primary key. Or if we have a list of elements belonging to some parent, they might be identified uniquely by the parent's ID and an integer sequence number reflecting their position in the list.

Keep in mind that a primary key means no two tuples can have the same values for that attribute (or those attributes) at the same point in time. Suppose we have a relation where the primary key is the parent's ID and the sequence number. So there are tuples like (92858, 0, Smith), (92858, 1, Kim), (92858, 2, Singh), where the first two elements form the primary key. Suppose we then delete the first element of the sequence; after adjustment we will have (92858, 0, Kim), (92858, 1, Singh). The primary key (92858, 0) refers now to a different tuple than it did before, but it is still unique at this point in time, so there is no problem.

If a relation references the primary key of another, we can make it a requirement that its content matches an entry in the domain of another table. Thus, we could make it a requirement that when a vehicle row is created, the license plate number must be one of the entries in the license plate table (or null, if that is permitted). Thus, if we tried to put in an entry that referenced license plate "DDDD 001" when no such plate exists in the system, this would be an error. This explicit constraint is called a *foreign key*.

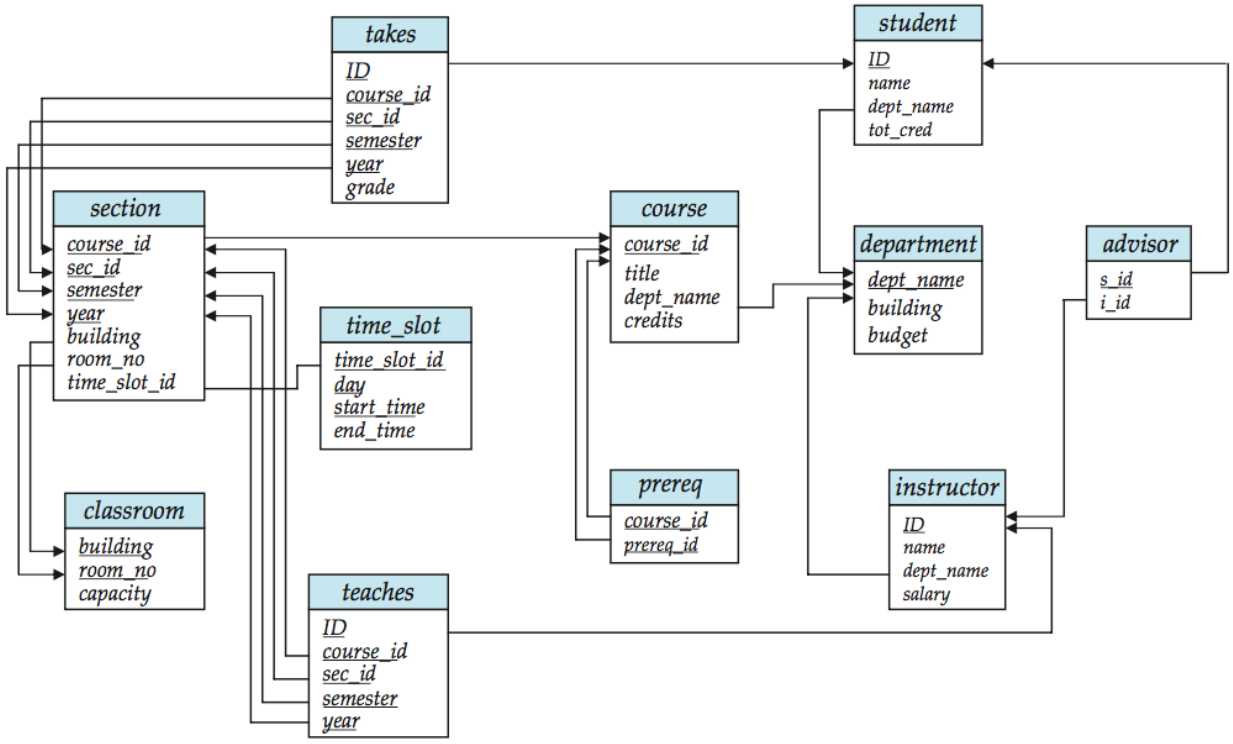
More formally, a foreign key is defined in [EN11] as:

1. The attributes of the foreign key of relation R_1 relating R_1 to relation R_2 have the same domain(s) as the primary key of R_2.
2. A value of a FK in a tuple t_1 of R_1 either occurs as a value of the primary key for some tuple t_2 in R_2 in the current state, or is null.

A foreign key is also called a referential integrity constraint: it ensures that a reference to another table is valid; that a referenced tuple does in fact exist in the target table.

Under best circumstances, the database server will reject any attempt to modify data such that violates referential integrity (or otherwise breaks a rule). If that is the case, then we may have some certainty that our data is in a valid state. If there are rules not entered into the system explicitly at the beginning it may be very painful to introduce them later if the system is in an invalid state.

Below is an example of a database schema that, in a very simple way, represents a university. In this diagram, each relation (table) is a box, with the name at the top in blue, and its attributes below. The arrows indicate foreign key dependencies.



A database schema diagram representing a "university" [SKS11].

3 — Relational Query Languages and SQL

Relational Query Languages

A *query language* is a language in which the user requests information from the database. Such languages can be procedural, which is to say that you tell the database exactly how to get the data you want, or non-procedural, in which case you just tell the database what you want and it figures out how to deliver that result[4].

The operations we want to discuss can be applied to a single relation, or a pair of relations; the result of is always a single relation [SKS11]. This means that the output of one operation can be used as the input to another operation, allowing us to chain operations as needed.

If it helps you to imagine this, every operation is a function with return type `Relation` and they take either one or two `Relations` as parameters. You will also need to keep in mind that because they all have input and output of relation(s), the operations operate on a set of data, not just a single element. In a C-like language you might write a for loop that sets all elements in an array to be `-1`. In SQL you would write a query that says set the value to be `-1` for all entries in the relation. This means that if your normal mode of thinking is C-like (procedural) then there is a mental transition that needs to be made.

The fundamental operations in relational algebra are [SKS11]:

- Select
- Project
- Union
- Set Difference
- Cartesian Product
- Rename

And there are three shortcuts we will use that can be created using the fundamental operations:

- Set Intersection
- Assignment
- Join

SQL – "Structured Query Language" – is not only for querying but also for defining and changing the database schema. It is a nonprocedural language: you ask for what you want and the database server returns a result that matches that format. What we will discuss in this class is SQL (and more specifically you will use a free implementation, MySQL, in the labs). As we go through the operations we will learn how to express the operation we want to do using SQL as a specific example. But other query languages do exist.

[4] There is of course the possibility that the database server chooses a SUPER INEFFICIENT way of carrying out the query, as we will discuss in the future...

The course notes and the way the material is examined reflects the reality that the majority of students taking this course already have exposure to some form of database system, most likely a SQL relational database of some sort. That may mean the material is fully review for the reader. Still, the mathematical notation may be new.

SQL consists of several parts [SKS11]:

- Data Definition Language (DDL) – Used to specify the schema, define domains, relations, integrity constraints.
- Data Manipulation Language (DML) – Used to query (read) as well as manipulate the data.
- Data Control Language (DCL) – Used to control access privileges in the database.
- Transaction Control Language (TCL) – Used to control the execution of transactions.

For now we will focus on the data manipulation language, although we will talk about the data definition language.

Let's use the following sample data to understand the operations.

VIN	year	make	model	license_plate_number
2B4FH25K1RR646348	2005	Honda	Civic	ZZZZ 249
4UZACLBW87CZ42980	2011	Ford	Focus	YYYY 995
WVWDM7AJ4BW227648	2015	Volkswagen	Golf	ZRHC 112
1J4FT38L7KL506678	2017	Audi	A4	VNTY PLT
2C8GF48415R447850	2016	Volkswagen	Jetta	VWJG 071
2T3RK4DV1AW089914	2014	Toyota	Camry	ZZZZ 385
1XPCDR9X2YN436206	2012	Toyota	Camry	ZZYY 251
1GYS3BKJ5FR338462	2016	Honda	Civic	YYYY 585
1GAGG25R6Y1243081	2015	Chevrolet	Corvette	GORLFAST
JH2SC59208M054959	2018	Genesis	Genesis	AAAA 123

Selection. The first operation we will discuss is selection. That is to say, find the tuples in the relation. As you might imagine, we need to specify what we want to find, and where we want to look for it. Also, we can specify that we want to find only the things that match certain criteria.

In mathematical notation, selection has the symbol σ (sigma). The predicate is put in a subscript and then the function parameter, the relation to select from, is follows in parenthesis. $\sigma_{make=''Volkswagen''}(vehicle)$ selects the tuples from the `vehicle` relation where the make is equal to "Volkswagen". That produces a new relation that looks like this:

VIN	year	make	model	license_plate_number
WVWDM7AJ4BW227648	2015	Volkswagen	Golf	ZRHC 112
2C8GF48415R447850	2016	Volkswagen	Jetta	VWJG 071

We specify the predicate p using propositional logic. A term is written as an attribute followed by a comparison operator then an attribute or constant. The comparison operators are $=, \neq, >, \geq, <, \leq$. So the attribute in the example is $make$ and the comparison operator is $=$ (equals) and then the right hand side of that equation is the constant "Volkswagen". Terms are connected by the mathematical AND $\wedge$, mathematical OR $\vee$, and mathematical NOT $\neg$ operators. Thus we could specify make = "Volkswagen" and year = "2015" ($\sigma_{make=''Volkswagen''\wedge year=''2015''}(vehicle)$) which would restrict the returned set of tuples to be those that match both parts of the predicate:

VIN	year	make	model	license_plate_number
WVWDM7AJ4BW227648	2015	Volkswagen	Golf	ZRHC 112

We can chain as many of these qualifiers as desired. Use of parenthesis is recommended to make it clear how the boolean logic is to be evaluated. If you wish the year to be either 2015 or 2016 and the make to be Volkswagen,

then it is advisable to write that as $make ='' Volkswagen'' \wedge (year ='' 2015'' \vee year ='' 2016'')$. My general rule is that it doesn't cost extra money to write the parenthesis so it is advisable to use them anywhere there is the possibility of error or confusion.

It is possible, of course, that no tuples in the relation match all of the predicates. If the predicate is make = "Chrysler" then no matches will be found in our sample data. Or, if we specify make = "Honda", and model = "Camry", we will still find no tuples that match that either. Honda doesn't manufacture any model named "Camry" so we expect not to find any such rows in the database, but Chrysler does make cars (still), so that data is plausible, it just so happens that we didn't find any in our sample data set. Either way we get back an empty relation.

In SQL, the `SELECT` clause is used to, well, select tuples from the database. The SQL command that corresponds to $\sigma(vehicle)$ is `SELECT * FROM VEHICLE;` Some observations spring from this immediately. For one thing, it is traditional in SQL that we write keywords in all capitals. Many databases don't really seem to mind if you use lowercase or mixed case, but some systems are case sensitive. Another item of note is that a statement is terminated by a semicolon (which is nothing new for those of us who write C-like languages). We also note the use of `FROM` to specify the relation. The asterisk (`*`) is needed to specify what we are going to select, a subject we will return to soon when we talk about projection.

The select statement as is contains no predicates so it would return all tuples in the relation. If we wished to add the predicate that make equals Volkswagen, the statement becomes `SELECT * FROM VEHICLE WHERE make = 'Volkswagen';` Some more observations then: the `WHERE` keyword indicates the start of the predicate. The string literal "Volkswagen" is enclosed in single quotation marks but it can also be in double quotation marks (although some database systems may object to this and insist on one or the other). To form a compound predicate: `SELECT * FROM VEHICLE WHERE make = 'Volkswagen' AND year = '2015';`.

The comparison operators are what you might expect for a programming language: `<`, `>`, `>=`, `<=` are all as expected in a C-like language. Equals is just a single equals sign (`=`) and the not equals operator is written `<>`, which although it looks really strange if you are a C programmer, is the correct not equals operator in BASIC. These operations can work on strings, mathematical types, dates, et cetera. You may, however, get unexpected results if you attempt to compare two types that don't compare well.

Although it is possible to chain the where clause to include a range, such as `WHERE year >= 2010 AND year <= 2015` there is a bit of notational convenience in SQL: the `BETWEEN ... AND` syntax. Thus one can write `WHERE year BETWEEN 2010 AND 2015`. This is inclusive on both ends, so be careful about whether this is the desired behaviour.

The predicate may also contain the name of other attributes in the relation, such as `SELECT * FROM VEHICLE WHERE make = model;`. This will return the one tuple in the sample data where make and model are the same. There are situations in real life where we want to use an attribute rather than a constant, although in this particular example it may look a little strange.

Projection. The second operation is projection. This takes a subset of a relation: take a relation and extract from it only the things that we asked for. Projection cuts down the relation to the specific attributes. Projection takes a single relation and returns another relation.

As you can imagine, this may help the user or application program: if we want to know what the VIN is for the car with the license plate "GORLFAST" we can ask for just that and need not get back the entire tuple. Similarly, if you want to get a list of the makes and models in the database, again, it's nice to be able to get this data without any of the parts you do not need. Projection also makes certain things a lot faster: it makes no sense to load all this extra data to your program if you are going to ignore almost all the columns[5].

In mathematical notation, projection has the symbol Π (capital pi). Like selection, it takes a subscript and the input relation follows in parenthesis. So, $\Pi_{make,model}(vehicle)$ will reduce the relation down to just the attributes make and model, leaving out VIN, year, and license plate number information. All tuples in the relation will appear in the output of the projection operation, although it is entirely possible that some of them look very much alike. See the example data below for when we project the data:

[5] At some point I will end up telling this story about how the SQL query optimizer is not very smart... Yes, this references the previous footnote...

make	model
Honda	Civic
Ford	Focus
Volkswagen	Golf
Audi	A4
Volkswagen	Jetta
Toyota	Camry
Chevrolet	Corvette
Genesis	Genesis

There are fewer tuples in this relation than there were in the original relation. This is because there are duplicates ("Honda"/"Civic", for example). In the mathematical world a relation is a set and therefore duplicates are not permitted. The order of the tuples is not specified either. The attributes will be listed in the order they are provided in the project clause, which we may choose arbitrarily. If there is a reason to have them in a certain order, then put them in that order. You can also ask for all attributes in the relation; this may simply re-order the attributes (columns) if you ask for all of them.

In SQL projection is done through our application of the `SELECT` clause. In the previous examples we used `*` which specified all attributes. In production the use of `SELECT *` is discouraged; usually the correct approach is to ask specifically for the things you want and discard all the things you don't want. So, the SQL equivalent of the previous mathematical operation is: `SELECT make, model FROM VEHICLE;` which returns:

make	model
Honda	Civic
Ford	Focus
Volkswagen	Golf
Audi	A4
Volkswagen	Jetta
Toyota	Camry
Toyota	Camry
Honda	Civic
Chevrolet	Corvette
Genesis	Genesis

This does not look the same as the previous output. What gives? Detection of duplicates is relatively difficult (computationally expensive). SQL does not remove them unless you ask, so technically what you get back is more akin to a list than to a set. You could also consider it a multiset. Nevertheless, if you wish to remove duplicates you may use the keyword `DISTINCT`: `SELECT DISTINCT make, model FROM VEHICLE;` which then eliminates duplicates from the results. The opposite of `DISTINCT` is the `ALL` keyword, but since this is the default behaviour you are unlikely to need to write it.

The SQL `SELECT` statement is then both selection and projection. In what order do these operations take place? Sometimes it does not matter, e.g., if we asked for `SELECT *...` then the projection does nothing. Here's a hint: this statement is perfectly valid: `SELECT make, model from VEHICLE WHERE year = '2016';` and returns:

make	model
Volkswagen	Jetta
Honda	Civic

Clearly the selection operation must take place first: if we cut the attributes down to just make and model, then we lose the information we need to tell whether the year is 2016 or not. Admittedly, when we get into the topic about how queries are carried out, we may find it is better to cut down the result relation with projection before applying the selection, then reducing it further. But let us not skip ahead too much.

The `SELECT` statement in SQL lets us do some interesting things other than select attributes alone. The select clause may contain arithmetic expressions using the operators $+, -, *, /$ operating on either attributes or tuples [SKS11].

Thus, to get a 2 digit date out of a 4 digit date we could do: `SELECT year - 2000 FROM VEHICLE;`. Assuming, of course, that the types are compatible. The utility of using the mathematical functions is perhaps somewhat limited for now, but there will be occasions in the future.

We can also select constants if we wish, by writing that constant instead of the name of the attribute. `SELECT 'Car', make, model FROM vehicle WHERE year = '2016';` produces:

Car	make	model
Car	Volkswagen	Jetta
Car	Honda	Civic

That maybe looks silly, asking for a constant value to be returned in a relation. There are some uses for the selection of a constant, as we will see shortly.

Union. We can combine two relations using the union operation. It takes two relations and produces a single relation from them.

The mathematical symbol is $\cup$, the same as the mathematical set union operator. Typically this is used in conjunction with other operations (selection, projection) to get the data that you want. The query $\sigma_{make=''Ford''}(vehicle) \cup \sigma_{make=''Audi''}(vehicle)$ produces the results:

VIN	year	make	model	license_plate_number
4UZACLBW87CZ42980	2011	Ford	Focus	YYYY 995
1J4FT38L7KL506678	2017	Audi	A4	VNTY PLT

The first selection statement returns a relation that contains one tuple. The second selection statement also returns a relation that contains one tuple. The union operator then takes those results and combines them into a single relation that contains both tuples. If we had asked for a projection we could either apply it to each selection operation, or to the result of the union.

In SQL the keyword is `UNION`. Use of parenthesis is recommended (if not mandatory) to help the SQL query parser figure out what it is you are asking for. `(SELECT license_plate_number FROM vehicle WHERE make = 'Ford') UNION (SELECT license_plate_number FROM vehicle WHERE make = 'Audi');`

But wait, you say, this is silly. Why did you not simply write $\sigma_{make=''Ford'' \vee make=''Audi''}(vehicle)$ (or its SQL equivalent)? It would produce the exact same results. And so indeed it would. The use of the union operation when selecting from a single relation is not wrong, but not necessary in this situation since we could use OR.

Where we will more often see union applied is when we want to combine data from two different relations. Suppose our database also has a relation (table) for the addresses of employees that has employee ID, name, street, city, province, and postal code attributes. Thus to get all the addresses in the system, the query in math notation is:

$(\Pi_{name,street,city,province,postal\_code}(\sigma(owner\_address))) \cup (\Pi_{name,street,city,province,postal\_code}(\sigma(employee)))$

In SQL:

```
(SELECT name, street, city, province, postal_code FROM owner_address)
UNION
(SELECT name, street, city, province, postal_code FROM employee);
```

The union operation can only succeed if the types are compatible. For the union operation to make sense, two conditions must hold [SKS11]:

1. The relations must have the same number of attributes.

2. The domain of attribute i in the first relation must be the same as the domain of attribute i in the second relation, for all i.

The previous example meets the criteria because we specified the attributes in the same order and they are presumed to be the same type, or at least, compatible domains.

The names do not have to be the same: if the employee relation called it "postcode" instead of "postal_code" then the union operation could succeed as long as they are both defined as the same type (e.g., string of length 7). Your SQL server might let you get away with things though: it might allow you to take the union of a license plate set and a postal code set because the domain of each of those is string. This makes no logical sense, but the database may let you do it. In some cases it will let you combine otherwise incompatible types if it can figure out a way to make it work... such as converting an integer to a string... but it is not recommended!

This example also is a situation where we might make use of a selection with a constant. If we wanted to produce a relation with all addresses, and wish to include some type information, we could do this:

```
(SELECT 'Vehicle Owner', name, street, city, province, postal_code FROM owner_address)
UNION
(SELECT 'MTO Employee', name, street, city, province, postal_code FROM employee);
```

	name	**street**	**city**	**province**	**postal_code**
Vehicle Owner	Jean Valjean	19 Rue des Prisonniers	Ottawa	ON	B1B 1B1
Vehicle Owner	Thomas Anderson	1234 Main St	Waterloo	ON	A0A 0A0
Vehicle Owner	Alice Jones	4 Generic Place	Kenora	ON	C2C 2C2
MTO Employee	Kim Morgan	491 Example Road	Waterloo	ON	N2N 2N2
MTO Employee	Jordan Singh	24 Eastern Avenue	London	ON	N6A 0B0

This combined result made use of projection, selection, and union.

4 — Relational Query Languages & SQL, Continued

Relational Query Languages, Continued

When we left off we had covered the operations of selection, projection and union. Now we will continue with set difference, cartesian product, and rename, followed by the shortcut operations.

Difference. The set difference operation works more or less as you would expect: it takes a relation and removes from it any tuples that are in the second relation. After taking away things we do not want, then we are left with the things we do want. Its input is therefore two relations and it produces a new relation as its output.

The mathematical symbol is $-$ (the minus or subtraction operator). The notation is then $r_1 - r_2$ where r_1 and r_2 are relations. This produces a relation r_3 that contains all tuples in r_1 that are not in r_2. It is possible to chain the subtraction operators, but like regular mathematical subtraction, it does not commute: the order matters a lot.

The query $\sigma_{make=''Honda''}(vehicle) - \sigma_{year<''2010''}(vehicle)$ produces the results:

VIN	year	make	model	license_plate_number
1GYS3BKJ5FR338462	2016	Honda	Civic	YYYY 585

Just like the union operation, the sets must be compatible. The same rules from union apply:

1. The relations must have the same number of attributes.
2. The domain of attribute i in the first relation must be the same as the domain of attribute i in the second relation, for all i.

In SQL the keyword we need for this is `EXCEPT`[6]. Like the union operation, parenthesis prevent confusion and/or make your database server less sad.

```
(SELECT make, model FROM vehicle WHERE make = 'Volkswagen')
EXCEPT
(SELECT make, model, FROM vehicle WHERE year < 2016);
```

Produces as output:

make	model
Volkswagen	Jetta

[6] Although Oracle uses `MINUS` instead.

Again, you say, wait, this is silly, you could cut this down with a select query with two clauses. Yes, that is also true. Those with experience in databases may also be asking about the use of `EXCEPT` vs another construct `NOT IN`. The `EXCEPT` keyword encompasses both the `DISTINCT` and `NOT IN` behaviour, so duplicates are eliminated. My typical use of the not-in behaviour usually refers to referencing a unique key column in the database, such as:

```
SELECT make, model FROM vehicle WHERE make = 'Volkswagen' AND VIN NOT IN
(SELECT VIN FROM vehicle WHERE year < 2016);
```

This demonstrates a subquery: the first query selects a relation with one attribute.

Cartesian Product. The cartesian product combines information from two relations. It is often the case that the data we need is in more than one relation. That might be a design problem, but good design means data is in there only once and sometimes we do need to combine relations. Suppose we wanted to send reminder notices to people whose license plates will expire next month. We need to combine license plate data with address data.

The mathematical symbol for cartesian product is $\times$, the multiplication symbol. The cartesian product $r_1 \times r_2$ forms a third relation r_3. The new relation has all the attributes of both the relations that went into it (in the order specified). An example would clarify.

Let us pretend our license plate relation contains exactly two tuples:

number	expiry	owner_address_id
ZZZZ 123	2018-09-30	24601
AAAA 855	2019-04-01	12949

And our owner address set is also two tuples:

id	name	street	city	province	postal_code
24601	Jean Valjean	19 Rue des Prisonniers	Ottawa	ON	B1B 1B1
12949	Alice Jones	4 Generic Place	Kenora	ON	C2C 2C2

Then the query $license\_plate \times owner\_address$ produces:

number	expiry	owner_address_id	id	name	street	city	province	postal_code
ZZZZ 123	2018-09-30	24601	24601	Jean Valjean	19 Rue des Prisonniers	Ottawa	ON	B1B 1B1
ZZZZ 123	2018-09-30	24601	12949	Alice Jones	4 Generic Place	Kenora	ON	C2C 2C2
AAAA 855	2019-04-01	12949	24601	Jean Valjean	19 Rue des Prisonniers	Ottawa	ON	B1B 1B1
AAAA 855	2019-04-01	12949	12949	Alice Jones	4 Generic Place	Kenora	ON	C2C 2C2

This is all possibilities (and that's why we restricted these relations to 2 entries, just so we don't waste too much space). Each tuple from the first relation is paired with each tuple from the second. So if r_1 contains x tuples and r_2 contains y tuples, there will be $x * y$ tuples in the resulting relation... the vast majority of which are garbage!

Garbage, you say? The second and third tuples in the relation shown immediately above don't make any sense: plate ZZZZ 123 is associated with the owner address with ID 24601, but it looks like it's also associated with address with ID 12949. Whom does the plate belong to, Jean Valjean or Alice Jones? Well, we know it's Valjean, but the cartesian product can suggest wrong things (and produces a very big relation). Even more illogical results arise if we try to take the cartesian product of one relation with itself.

This problem did not occur with our particular example because all attribute names are distinct, but if we had two names that were the same, we would need a way to differentiate them. Suppose that two relations, one called `book` and one called `author`, each having an attribute `id`. The convention is to prefix the attribute with the name of the relation from which it came. Thus, in the cartesian product the two columns would be `book.id` and `reader.id`.

In SQL it is very simple to get the cartesian product: `SELECT * FROM license_plate, owner_address;` – that is to say we just separate the relations we wish to appear in the cartesian product with a comma in between.

We can always combine more if we needed: imagine we need to notify all current owners of Volkswagen vehicles that there is a recall: this requires us to look at both the vehicle relation and the owner address relation. To connect those we also need the license plate relation (this escalated quickly).

Typically get our data to make sense we need to restrict our query with a selection predicate. If the two relations are connected by some ID in in common, we need that. In this case, the `id` attribute in the address matches `owner_address_id` in the license plate relation. So $\sigma_{owner\_address\_id=id}(license\_plate \times owner\_address)$. Or in SQL: `SELECT * FROM license_plate, owner_address where owner_address_id = id;` This may seem unclear so we might prefer to prefix these with their names: `SELECT * FROM license_plate, owner_address where license_plate.owner_address_id = owner_address.id;`

If we add that restriction:

number	expiry	owner_address_id	id	name	street	city	province	postal_code
ZZZZ 123	2018-09-30	24601	24601	Jean Valjean	19 Rue des Prisonniers	Ottawa	ON	B1B 1B1
AAAA 855	2019-04-01	12949	12949	Alice Jones	4 Generic Place	Kenora	ON	C2C 2C2

Now our data makes sense! In the future we will get to the same place through an easier operation, the join.

Rename. The rename operation is used to both change the name of existing attributes/relations and assign names to ones that have no name.

The mathematical notation is ρ (rho). If applied to a relation, it is $\rho_x(E)$ where it renames the relation to x. Or it can be applied to the name of the attributes: $\rho_{x(a_1,a_2,a_3,a_4...)}(E)$ which renames the relation to x and then the attributes a_1...

In SQL we can use the `AS` keyword. That is not super interesting on a simple query to rename a relation: `SELECT * FROM vehicle AS autos;` renames the relation result to "autos" but that does not do very much for us. It makes some more sense if we are going to use it in a subquery it will change the prefix for any duplicate attribute names (e.g., change `book.id` to `textbook.id`).

A more likely use is to rename the attributes of a relation. We can apply it to as few or as many attributes as we like: `SELECT make, model, year AS modelYear FROM vehicle;` will produce a relation with the make, model, and year, but the year attribute will be called "modelYear" as we have asked.

The motivation for the rename operation right now probably seems weak. There are some situations where we will need to use the rename operation. And I do mean need, when we do not have an AS clause on certain operations the SQL server will decline to carry out the query.

Additional Operations

Having reached the end of the basic operations, there are a few operations that we have discussed that are not fundamental: they can be derived from the six operations we have already introduced. They are, however, notationally convenient. These are from [SKS11]:

Set Intersection. Set intersection is exactly what it sounds like based on our understanding of mathematical sets. The symbol for it is $\cap$ and it is equivalent to $r_1 - (r_1 - r_2)$.

In SQL they keyword is `INTERSECT`. Its use is limited, however, in a way similar to union. If we want to use intersection on two queries of the same relation, we could just as easily set it up as a selection with a compound predicate. But a short example:

```
(SELECT name, street, city, province, postal_code FROM owner_address)
INTERSECT
(SELECT name, street, city, province, postal_code FROM employee);
```

This would produce the set of addresses that are both in the owner address relation and in the employee relation.

Assignment. The assignment operation allows us to take the result of an expression and put it in a temporary variable. The mathematical symbol for this is $\leftarrow$. This is just notational convenience for the benefit of the reader. Instead of a complicated expression that requires several sets of parenthesis and is hard to follow, we could break up the query into parts and then use those parts. So instead of something $\sigma_{owner\_address\_id=id}(license\_plate \times owner\_address)$, we could write:

$temp \leftarrow license\_plate \times owner\_address$
$\sigma_{owner\_address\_id=id}(temp)$.

This may improve (or decrease) readability if used well (or badly).

SQL does have an assignment operator: `:=`. I will, however, discourage you from using it. There are advanced situations where the use of this is appropriate, such as manipulating a counter. Eventually we will learn how to modify the data in our database and not just look at it and when we do, an assignment will take place, but we are unlikely to write it explicitly. The reason I discourage the use of assignment is the need to get away from the C-like thinking with counters and iteration; operations take place on a set of operations and we should think in a set based mindset.

Join. We have already seen the cartesian product operation tends to produce a lot of rows, some of which are illogical. By combining that with a restriction, i.e., a selection, we throw away the tuples that don't make sense and are left with only those where there is some meaning. One is hard pressed to think of a situation where this is not needed. Thus, a little notational shorthand is in order.

The mathematical symbol for the *natural join* is $\bowtie$ (bowtie). It combines the cartesian product with a selection, forcing equality. A natural join combined with a selection is a theta join $\bowtie_\theta$ where θ is the selection predicate. Note that it is associative: so we can chain it as we need: $vehicle \bowtie license\_plate \bowtie owner\_address$ is equivalent to $(vehicle \bowtie license\_plate) \bowtie owner\_address$ and $vehicle \bowtie (license\_plate \bowtie owner\_address)$ [SKS11].

Except, this probably doesn't work the way we hope on our sample data because the fields are not named the same thing. If both fields are called `id`, the database server will try to match on those. But this is not going to work for our data since they all have different names.

In SQL the keyword we need for this is `NATURAL JOIN`. So we would do: `SELECT * FROM vehicle NATURAL JOIN license_plate;` Again, though, this is not going to cooperate for us because our names don't match.

What do we do then? We need to expand our concept of the join operation. The default type is the `INNER JOIN` and it's what you get if you just write `JOIN` instead of being specific. It requires us to use the `ON` clause to specify how we relate one side to the other.

`SELECT * FROM license_plate JOIN owner_address ON owner_address_id = id;` produces:

number	expiry	owner_address_id	id	name	street	city	province	postal_code
ZZZZ 123	2018-09-30	24601	24601	Jean Valjean	19 Rue des Prisonniers	Ottawa	ON	B1B 1B1
AAAA 855	2019-04-01	12949	12949	Alice Jones	4 Generic Place	Kenora	ON	C2C 2C2

My general recommendation is always to use the `ON` clause rather than `NATURAL JOIN`. Oftentimes you need it and when you don't you sometimes get the wrong behaviour because a natural join takes the unique ID field from each relation and not the field that actually links them. If the license plate relation called the plate number "id" instead of "number", the natural join operation would run but produce no results. Be explicit about how the join should work.

If there is an inner join, there is obviously an outer join. Actually, there are three of them! They are, however, all very similar.

Suppose we have a license plate that does not correspond to any address:

number	expiry	owner_address_id
ZZZZ 123	2018-09-30	24601
AAAA 855	2019-04-01	12949
BBCC 394	2019-02-12	null

and an address that does not correspond to any license plate:

id	name	street	city	province	postal_code
24601	Jean Valjean	19 Rue des Prisonniers	Ottawa	ON	B1B 1B1
12949	Alice Jones	4 Generic Place	Kenora	ON	C2C 2C2
86753	Sherlock Holmes	221B Baker St	London	ON	D4D 4D4

The inner join of license plate with owner address would look like this:

number	expiry	owner_address_id	id	name	street	city	province	postal_code
ZZZZ 123	2018-09-30	24601	24601	Jean Valjean	19 Rue des Prisonniers	Ottawa	ON	B1B 1B1
AAAA 855	2019-04-01	12949	12949	Alice Jones	4 Generic Place	Kenora	ON	C2C 2C2

... that is to say, no different from how it was before. Because there is no match for license plate "BBCC 394" it does not appear in the output relation when the natural join is used. The same applies to address with ID "86753"; it has no match in the license plate relation.

If we use the left outer join, which has the symbol ⟕, we get a tuple in the result for each entry in the left (first) table, even if it has no match in the second. For such a tuple, all attributes from the right (second) table will simply be "null". If a tuple in the right table has no match in the left table, it does not appear at all. The SQL for this is, of course, `LEFT OUTER JOIN`. So `SELECT * FROM license_plate LEFT OUTER JOIN owner_address ON owner_address_id = id;` returns:

number	expiry	owner_address_id	id	name	street	city	province	postal_code
ZZZZ 123	2018-09-30	24601	24601	Jean Valjean	19 Rue des Prisonniers	Ottawa	ON	B1B 1B1
AAAA 855	2019-04-01	12949	12949	Alice Jones	4 Generic Place	Kenora	ON	C2C 2C2
BBCC 394	2019-02-12	null	null	null	null	null	null	null

The right outer join, symbol, ⟖ (`RIGHT OUTER JOIN`) is the mirror image operation of the left outer join. We get a tuple in the result for each entry in the right (second) table, even if it has no match in the first. For such a tuple, all attributes from the left (first) table will simply be "null". If a tuple in the left table has no match in the right table, it does not appear at all.`SELECT * FROM license_plate RIGHT OUTER JOIN owner_address ON owner_address_id = id;` returns:

number	expiry	owner_address_id	id	name	street	city	province	postal_code
ZZZZ 123	2018-09-30	24601	24601	Jean Valjean	19 Rue des Prisonniers	Ottawa	ON	B1B 1B1
AAAA 855	2019-04-01	12949	12949	Alice Jones	4 Generic Place	Kenora	ON	C2C 2C2
null	null	null	86753	Sherlock Holmes	221B Baker St	London	ON	D4D 4D4

In both the left and right outer join, it is important to note the order in which the relations appear. The operation thinks of the left and right side from the perspective of the placement of the operator (as one would expect).

Finally, there is the full outer join, with the symbol ⟗ does both the left and right outer join operations, including tuples from each relation that do not match the other:

number	expiry	owner_address_id	id	name	street	city	province	postal_code
ZZZZ 123	2018-09-30	24601	24601	Jean Valjean	19 Rue des Prisonniers	Ottawa	ON	B1B 1B1
AAAA 855	2019-04-01	12949	12949	Alice Jones	4 Generic Place	Kenora	ON	C2C 2C2
BBCC 394	2019-02-12	null	null	null	null	null	null	null
null	null	null	86753	Sherlock Holmes	221B Baker St	London	ON	D4D 4D4

It might be tempting to combine tuples from one relation with those from another where there are suitable null attributes, but this is misleading: the license plate BBCC 394 is NOT associated with the address of Sherlock Holmes and it does not make sense to combine these.

5 — More SQL

SQL: the Sequel

With some theoretical understanding of the relational algebra that makes up the basis for SQL we can now take some time to learn a little more of the practical ins and outs of the language that will allow us to do the things we need our database to do.

String Operations. String operations are complicated in any language and SQL is no exception. We have used some string matching already so we have a pretty good idea how it works: by default a string is enclosed in single-quote characters such as `'AAAA 111'`. Double quotes can also be used, especially if you need to enclose a single-quote literal in your string such as: `"Burk's Falls"`.

The SQL standard says that string comparison with = is case sensitive, but MySQL (and some others) do not respect this and perform case-insensitive comparison [SKS11]. Standards are optional, I guess? MySQL has gotten better and made this a configurable parameter, but it's still something to be wary of.

Some functions familiar from C-like languages exist: `UPPER()`, `LOWER()`, `TRIM()`, et cetera. Pattern matching can be done using `LIKE` which takes two special characters as wildcards: `%` (match any substring) and `_` (match any single character) [SKS11]. If you need those characters explicitly, they are escaped using the `\` as one might expect. The backslash itself can also be escaped using the backslash.

Some examples might clarify how it works: the clause `LIKE 'Marc%'` would match the strings "Marc", "Marco", "Marcelline", and so on. The clause `LIKE 'Marc_'` would match only "Marco" from that previous set (exactly one character is required).

Null Values. If you have programmed with Java or C# or similar then you probably know that the existence of `null` sometimes complicates your life (`NullPointerException` anyone?). A null attribute indicates a lack of a value and it causes a certain chaos with some of our operations.

Arithmetic expressions involving null (e.g., addition, subtraction, etc) always results in null if any one of the operands is null. Comparisons involving null are also a hassle; saying whether 1 is "less than" null is not sensible so the value will be "unknown" [SKS11].

Use of "unknown" values in boolean operators is also complicated [SKS11]:

- AND: Comparing true AND unknown results in unknown; false AND unknown results in false; unknown AND unknown results in unknown.
- OR: Comparing true OR unknown results in true; false OR unknown results in unknown; unknown OR unknown results in unknown.
- NOT unknown also results in unknown.

If a where clause for a particular tuple evaluates to false, it is not added to the tuples to be returned or changed.

It is possible, though, to test whether something is null, or is not null. As you might expect, those are `IS NULL` and `IS NOT NULL`. Example: `SELECT * from VEHICLE WHERE year IS NULL;` finds all tuples in the vehicle relation where the year attribute is null. It is correct to use the `IS NULL` or `IS NOT NULL` constructions rather than something like `= null` or `<> null`.

Ordering. Thus far if multiple tuples are returned by a query, we get them in no specified order. You may see consistent behaviour across repeated queries, but no promises are made about the order in which they are delivered. We can change that, if we want, through the use of an `ORDER BY` clause. We specify the attribute to sort by, and we can choose ascending (`ASC`) or descending (`DESC`). An example: `SELECT * FROM VEHICLE ORDER BY make ASC`.

We may specify multiple attributes in the `ORDER BY` clause, such as `ORDER BY make DESC, model DESC`. This sorts by the "make" attribute (descending), and if the values of two tuples are equal in that attribute, then by "model" (descending). If no two tuples have the same first sort attribute then the second sort does nothing (why should it?). And you can, if desired, choose divergent directions: mixing ascending and descending. They are always handled from left to right so they cannot conflict.

The field used to order the tuples does not have to be returned in the result set you have selected. The order will still be imposed but it may not be obvious from looking at the result relation what the ordering actually is.

Aggregate Functions. Aggregate functions perform a reduction on the data. Reduction is something you've done a lot of, most likely, without knowing it: if you are given an array of integers and asked to sum it up, that's a reduction. You condense an array of values into a single one. That's a subject that gets some more attention in ECE 459: Programming for Performance. But the database can do some of these things for us!

The aggregate functions in SQL mostly are [SKS11]:

1. `AVG`: Average
2. `MAX`: Maximum
3. `MIN`: Minimum
4. `SUM`: Sum (total)
5. `COUNT`: Count (obviously)

These work more or less like you would expect: you can `SELECT SUM( salary ) FROM employee;` to get the total salary of all employees, or `SELECT AVG( salary ) FROM employee;` to get the average salary of all employees. Maximum and minimum are also pretty self-explanatory. Sum and average work only on numbers, but non-numeric types can be used for maximum and minimum.

`SELECT COUNT( salary ) FROM employee;` tells you the number of entries in the table where there is a salary. More commonly you would see `SELECT COUNT(*)` to count the number of entries in the relation. You could also put the primary key in the parenthesis instead of the asterisk if the asterisk gives you worries about performance nightmares.

One can also put `DISTINCT` in the count expression. If you wished to count the number of distinct model names, you could write `SELECT COUNT( DISTINCT make ) FROM vehicle;` which would remove duplicates.

According to [SKS11] SQL does not permit combination of `DISTINCT` and `COUNT(*)`. Although it seems like this is accepted by MySQL, it is nonstandard behaviour if it does work.

It is nice to be able to sum up or average over all tuples matching some clause, but we can get grouping if we want as well. It might be interesting to know there are 150 different models of car, and if we are curious we can add a where clause to find out how many are made by Audi, and then another one where we find out how many are made by BMW... But that is inefficient. The solution is grouping:

Example: `SELECT make, COUNT(model) FROM vehicle GROUP BY make;` might produce a result like the table shown below:

make	count(model)
Honda	1
Ford	1
Volkswagen	2
Totoya	1
Chevrolet	1
Genesis	1

One may of course use the `AS` keyword to rename the count attribute if desired or restrict the query with a where clause. We could group by more than one criterion, such as grouping by both make and model, in which case we would could get a count of the number of Honda Civics on the database, for example. Grouping works as you would generally expect.

The select statement, obviously, selects over some attributes of the relation and they must be either (1) in the group by clause or (2) aggregated in some way. The statement `SELECT make, COUNT(model), year FROM vehicle GROUP BY make;` is invalid because the attribute "year" is not in the group by clause nor is it aggregated. This is sensible: there are multiple values for the "year" attribute and we would have no way of knowing which is the correct value to output here.

That's nice, but what if we wanted to return only those rows where a certain property holds, such as count is greater than 1? For that there is the `HAVING` clause which would be applied as follows: `SELECT make, COUNT(model) FROM vehicle GROUP BY make HAVING COUNT(model) > 1;` This would return only the one tuple (Volkswagen).

The textbook [SKS11] gives the following evaluation order for such a query:

1. The `FROM` clause is evaluated to get the relation.
2. The `WHERE` clauses is evaluated on the tuples in the relation specified in the `FROM` clause.
3. Tuples matching the `WHERE` clause are placed into groups according to the `GROUP BY` clause (if this is absent, then it's all one big group).
4. The `HAVING` clause, if any, is evaluated and removes groups that don't meet the having condition.
5. The `SELECT` operation is performed, producing the tuple for each group.

In general, aggregate operations on null values tend to ignore any nulls they encounter: if the SQL query sums salaries and there is a null value it will not affect the sum (i.e. it will be treated like zero). The `COUNT` function does not, however. As usual, nulls have the potential to cause chaos.

Subqueries & Set Comparison. We can have subqueries (nested queries) where the result of one query is used as input to another query in a single statement. The keyword we are going to use extensively here is `IN`.

The first use of a set comparison we might think of involves replacing a query that looks like this: `SELECT * FROM employee WHERE id = 2419 OR id = 2917 OR id = 4058 OR id = 5911;` with: `SELECT * FROM employee WHERE id IN (2419, 2917, 4058, 5911);`. This is more compact and easier to read; in a simple example like this there is not much possibility of confusion of the OR clauses, but in a query with multiple parts joined by AND and OR clauses, parenthesis are soon needed to avoid confusion.

Instead of an explicit list, the list could be generated by a query. Suppose the ministry of transport has discovered that license plates in the range CCCC 001 through CCCC 999 were manufactured incorrectly and are prone to falling apart easily. The ministry can send an e-mail to everyone affected to tell them of the problem and offer a free replacement. `SELECT name, street, city, province, postal_code FROM owner_address WHERE id IN (SELECT owner_address_id FROM license_plate WHERE number LIKE 'CCCC%');`

Let us pretend that an auto manufacturer has a recall, say, they faked the emissions tests data. The ministry of transport could send letters to everyone who owns such a vehicle to let them know that if they do not get the dealer repair they will be unable to renew their license plates. In that case we need to have three tables linked and we can do that with subqueries.

```
SELECT name, street, city, province, postal_code
FROM owner_address WHERE id IN
  (SELECT id FROM license_plate WHERE number IN
    (SELECT license_plate_number FROM vehicle WHERE make = 'Volkswagen' AND model = 'Golf')
  );
```

We can usually replace a subquery with a join query, and vice versa. In many cases it is a question of taste, but sometimes a subquery is easier to read and understand (especially if it is complex).

Other Things. This has by no means been an exhaustive list of all the SQL keywords one could possibly use (e.g., `UNIQUE`, `WITH`, et cetera). We may introduce additional syntax later if it is relevant to some other operation. For the moment, however, we have enough syntax covered to continue on to modification as well as design.

Modification

Until now all the operations have been of the look-but-don't-touch variety – we have looked at the data, combined it, sliced and diced it, and generated some temporary values – but that's it; we haven't actually changed the data in any permanent way. There are three basic operations we can do: add, remove, and change.

Keep in mind that modification operations are transactions (that is to say, they are supposed to succeed or be as if it never started). So what really happens is that a new relation is created and that replaces the original relation. These operations can also be expressed using relational algebra with an assignment.

Insert. The insert statement creates a new tuple and adds it to the relation, or creates a set of tuples and then adds that set to the relation.

If we are creating just one, then the simple, manual approach is sufficient. In SQL the keywords are `INSERT INTO` and `VALUES`. To add a new license plate we would write: `INSERT INTO license_plate( number, expiry, owner_address_id ) VALUES( 'HOLMES01', '2018-12-10', 86753 );`

Observations: we specify the relation and then we give a list of attributes that we want to assign, followed by the values that we wish to be assigned. In some situations we can leave off the parenthesis-enclosed list of attributes but that is not recommended. The SQL server will try to carry out the assignment of values in the order that the attributes are defined in the database and this may result in undesired behaviour. I therefore insist that the use of the order of attributes is important and you should use it everywhere; doing less is just bad practice.

Furthermore, we can put the attributes to be inserted in any order as long as the value types match: `INSERT INTO license_plate( expiry, number, owner_address_id ) VALUES( '2018-12-10', 'HOLMES01', 86753 );` is completely equivalent to the previous statement.

It is permissible to leave some attributes out of the insert statement. `INSERT INTO license_plate( expiry, number ) VALUES( '2018-12-10', 'HOLMES01' );` would create a tuple where the owner address id attribute contains null. This must, however, be permitted by the table definition: i.e., the field must allow null attributes or have a defined default (a boolean attribute, for example, may have "default false" as part of its definition).

We may also insert multiple elements into a relation based on the result of a select statement. `INSERT INTO owner_address ( SELECT * FROM employee_address ) );` is a minimal statement that takes the result of the selection and uses it as a set of tuples to insert. This statement leaves off the attribute specification (which is possible, but not recommended) and has no predicate (where clauses) so it is a rather dangerous statement to write. These sorts of statements are hopefully rare occurrences in your database because they have a risk of going wrong or duplicating data. But sometimes, when, for example, two tables are being merged, they are unavoidable.

In relational algebra, if the relation is r and the new tuples are t, then the insert statement is $r \leftarrow (r \cup t)$.

Delete. The delete operation can only be used to remove a set of tuples from a single relation; it does not alter attributes in the tuples [SKS11]. If the goal is to "clear fields" the update statement is appropriate. A delete statement cannot be used on multiple relations at once.

The keyword in SQL is `DELETE`[7] and the statement `DELETE FROM license_plate WHERE number = 'BBCC 394';` will remove from the license plate relation any and all tuples that match the predicate. If no predicate is specified, then ALL tuples in the relation are removed (yikes!).

Like an insertion we may use the result of a subquery as the predicate in a where clause, so we could delete all license plates where the registered address province is not Ontario with something like `DELETE FROM license_plate WHERE owner_address_id IN (SELECT id FROM owner_address WHERE province <> 'ON');`

The textbook [SKS11] has an example about deletion where the salary is less than the average. This highlights that the deletion operation first figures out the average, then decides what tuples to delete, before carrying out the operation. If this were not the case, after each deletion the average might change and we might delete all tuples (or the result would otherwise be strange). If it carried out the deletion in such a manner it would also break the rule we have about this being a transaction: the tuples to be deleted should be removed in one swift stroke with no partially completed state visible.

In relational algebra then the relation r is modified by a deletion with predicate p as follows: $r \leftarrow (r - \sigma_p(r))$.

Deletion should be used very carefully. When data is removed from the database, it's gone and not coming back (except, well, from backups... You do take backups, right?). Sometimes rather than actually deleting things you may be well advised to "deactivate" them, whether by having a boolean attribute for "deactivated" or having a second relation as a sort of "recycle bin"/"trash can" where tuples to be removed will eventually go.

Update. The update statement changes one or more attributes of the tuple without changing all the values in the tuple. The update statement, like the delete operation, operates on a set of tuples: one may specify the predicate with the `WHERE` clause just as before.

In SQL the keywords we need are `UPDATE` and `SET`. To update someone's address: `UPDATE OWNER_ADDRESS SET postal_code = 'B1B 2B2‘ WHERE id = '24601';.`

By now there should be nothing unexpected in this statement: we specify what fields to assign, their new values, and we limit the tuples to be affected by use of the `WHERE` clause and its predicate.

It is necessary to specify at least one attribute to change, although it is possible that no tuples are changed because none match the where-predicate. Similarly, an update may fail if we try to violate one of the rules: if the new text is too long for the attribute, for example. There are also some rules about updating an identifier for this relation.

In [SKS11] there is an example involving salaries where the statement runs along the following lines: `UPDATE employee SET salary = salary * 1.50 WHERE id = 1;.` This takes the current value of salary for the employee with id "1" and increases it by 50% (the CEO is ever so generous to the CEO, no?). In this case the attribute salary is both read and written. It is perfectly alright to use and update a value like this, in much the same way that it is permissible to write a statement like `x = x + 5;` in a C-like language; the assignment takes place after the right hand side expression has been evaluated.

Similarly, if the statement is `UPDATE employee SET salary = salary * 1.05 WHERE salary > 100000;`, the behaviour is like deletion in that the set of salaries to update is determined before any modifications begin. The whole statement, no matter how many tuples it affects, is viewed as a transaction.

We could model this in relational calculus as simultaneous deletion of the old tuple and insertion of the new one.

[7] `https://www.youtube.com/watch?v=4ecWDo-HpbE`

6 — Data Definition

We have thus far not yet learned about how to formally create relations. It is now time to fix that. The SQL data definition language (DDL) looks a lot like the query language that we have used thus far, but it allows us to define the structure of the data.

Specifically, our data definition language allows us to define [SKS11]:

- The schema of each relation.
- The type of each attribute.
- Integrity constraints.
- Indices on relations.
- Security/authorization information for a relation.
- Physical storage structure.

Attributes have types, and the SQL standard includes the following built in types [SKS11, EN11]:

- **char**(n): A fixed length character string with a user-specified length of n. This one should never be used because it pads strings and it means comparisons are a pain and comparing a char attribute with another kind of string is an issue.
- **varchar**(n): A variable length character string with user-specified length n.
- **int** (or **integer**): an integer (exact size and maximum value is system dependent).
- **smallint**: a smaller integer. A fun-sized integer, so to speak.
- **numeric**(p, d): a fixed-point number with user-specified prevision; p is the the number of digits and d of those are to the right of the decimal point.
- **real**: floating point with a machine-dependent precision.
- **double precision**: double-precision floating point with machine-dependent precision.
- **float**(n): a floating point number with precision of at least n.
- **boolean**: A boolean value (what did you expect?)
- **date**, **time**, **datetime**: self explanatory, right?

Schema Definition - Create Table

If we wish to define a SQL relation, the syntax for this is to create a relation (table) is called (unsurprisingly), `CREATE TABLE`. The syntax for this command requires a name as well as a listing of the attributes (fields) and their types (definitions). It is also customary to include at least one constraint, the primary key. As before we put a semicolon at the end of the statement to designate the end of the statement.

```
CREATE TABLE r
 (A1 D1, A2 D2, ... An Dn,
 integrity-constraint-1,
 ...
  integrity-constraint-k);
```

A more concrete example:

```
CREATE TABLE student
 (id varchar(8),
  userid varchar(8) NOT NULL,
  firstname varchar(64),
  lastname varchar(64),
  birthday date,
  department_id int default 0,
  PRIMARY KEY( id ),
  FOREIGN KEY( department_id ) REFERENCES department( id )
 );
```

On some attributes an additional qualifier `not null` was added, and this means that a value of `null` is forbidden from being assigned to that attribute. This is, in a way, a form of integrity constraint. Like all integrity constraints, it means an insert or update statement that tries to set such an attribute to an impermissible value will be rejected.

The department ID attribute has a default value; if an insert statement does not specify the value for that attribute, rather than leaving it null, it will be the default value (here, 0) instead. A common case where not null and a default value could be combined is a boolean value, where the field may not be null and is by default "false".

The primary-key definition in this case specifies that `id` is the primary key for the relation which imposes requirements that the attribute cannot be null and must also be unique. In this situation we have defined the primary key as exactly one attribute, but a relation's primary key may be formed by the combination of several attributes.

The last sort of constraint that is shown in the example is the foreign key constraint that mentions the department ID. It means that for each student, the department id value must either be null or the value must match to the id value of a department tuple. If the types of the attributes don't match (e.g., department is defined with varchar for its id rather than integer) then adding this foreign key constraint will fail.

It is not shown in the diagram but we can override the default behaviour for what happens if the foreign key constraint is violated. Rather than rejecting the update we could choose if we want (1) to cascade the changes (alter the content of another table to make sure the rules are followed), or (2) setting null if the value is invalid, or (3) setting some default value. My personal preference is that we stick with rejection: it is better to prevent insertion of wrong data (and fix it at the source) rather than let it proceed and cover it up by putting a null in there. In the words of my friend Tuomo Jorri, if the date calculation is consistently off by 42 days, instead of a statement that says (date = date - 42), you should figure out why the date is off by 42 and fix that.

Not shown in the example above is the unique constraint: this requires simply that no two tuples of the relation may be equal in a particular attribute. Because null is not equal to null in the SQL standard, many tuples may have null for this value.

It is possible (although rare, at least in my experience) to also add a *check* clause as a constraint. The check clause takes a predicate and any insertion or update is is evaluated to see if it is consistent with this constraint. A check that says *salary > 0* ensures that an employee cannot be put into the database with a negative salary amount, for example [SKS11]. A check constraint predicate can be arbitrarily restrictive, allowing business logic to be embedded in the database (although it is unclear if this is desirable or not).

We did not do this in the above case, but we can put a name to foreign keys or other constraints. The name must be unique in the database schema. We'll look at adding names when we talk about altering tables. Names can be useful though, because a well-named constraint can tell you what has gone wrong: if its name tells you what relations are being referenced (e.g., the key indicates it links students with departments) then if the constraint is violated on an insert or update then you know immediately what rule was being broken.

Another way that adding a foreign key might fail is if the target relation does not exist. This means we would have to create relations in some order that means the foreign keys are all satisfied at the time of the creation. That might not be realistic, though, based on the desired schema. Fortunately, we can add them in later.

Altering tables

In addition to adding in some integrity constraints we can change the table definition, or remove constraints. The command for this is `ALTER TABLE` and we will need to specify the table to be modified as well as the change that we would like to make.

If we want to add a new column to the table, then the alter table syntax requires us to specify the name of the new attribute to be added and the type. `ALTER TABLE students ADD COLUMN email VARCHAR(128);` Subsequent to that we can make additional changes, such as adding an index, or putting in a reference constraint (which we will see how to do soon). At the time of creation, we can set a default value or set an attribute to be not null, however. Note that is is possible to add multiple elements in a single statement.

The mirror operation to that is to drop a column: `ALTER TABLE students DROP COLUMN email;` This deletes the attribute from the database and all the data that is in there, permanently. Dropping the column can fail if the column is used in a foreign key or other constraint. It may also be necessary to remove an index...

If we wish to rename a table, the keyword we need is `RENAME`: `ALTER TABLE students RENAME users;` This would change the name of the students relation to be `users`. It is not very exciting.

To change the definition of a column, we can use `MODIFY` or `CHANGE` depending on what we want to do: `ALTER TABLE users MODIFY email VARCHAR(255);` changes the definition of the email attribute; if we use `CHANGE` we have to specify the new definition including the name.

To add an index to a table, we (may) need a name for the index, and specify the relation and attribute(s) it should be created on: `CREATE INDEX idx_lastname ON users (lastname);`. But we don't have to, we could use a slightly different syntax: `ALTER TABLE users ADD INDEX (lastname);` which creates an index without needing to give it a name. In truth, it will be given a default name, but you at least don't need to make one up. Management of an index is not specified in the SQL standard so there's a fair amount of difference between the various vendors.

To remove an index, `ALTER TABLE users DROP INDEX lastname;`. If we didn't explicitly give the index a name it is likely the name of the column itself. If we really have no idea what the name of an index is, we can use `SHOW INDEX FROM users;` to ask the server to give us the information we need.

To add a foreign key, it gets complex: we need to define both tables first. Suppose there is a login session for an application and it is associated with a user's id attribute. To add a constraint we need to:

```
ALTER TABLE sessions
ADD CONSTRAINT FK_SESSION_USER
FOREIGN KEY (userid) REFERENCES users(id);
```

There is alternative syntax along the lines of `ADD FOREIGN KEY` but as a design decision I discourage it. In the above example the foreign key gets a name `FK_SESSION_USER` which is helpful in debugging. Adding the foreign key constraint in this way will fail if the two attributes (userid and id) do not have the same domain (e.g., they are not both varchar with length 8). Worse than that, if one of the tables is defined in one character encoding (UTF-8) and the other is defined in a different encoding (UTF-8mb4) then adding the constraint will fail and it might seem like a mystery as to why!

To remove a foreign key: `ALTER TABLE users DROP FOREIGN KEY FK_SESSION_USER;`

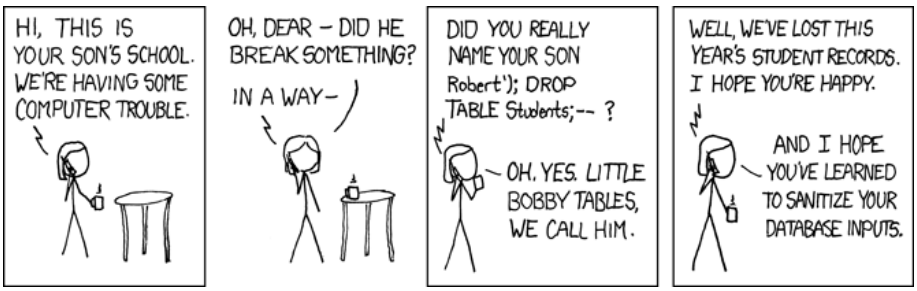
Truncate Table

If we wish to remove all tuples from a relation without affecting the structure at all, the command for that is to truncate the table: `TRUNCATE TABLE students;` would remove from the database all tuples of the students relation but would leave its definition unchanged. Truncating the table may fail if it would violate some constraints (breaking referential integrity).

There are some scenarios where this is a desirable operation. One possible scenario is if you have some application where login session information is stored in the database and you want to invalidate and remove all sessions you can truncate the table.

Drop Table

If we wish to remove a relation from the schema, the syntax for this is `DROP TABLE students`. This deletes the table and all of its content and then the content is permanently lost. And again, the drop operation may fail if the table to be deleted is referenced in some external constraints.



Obligatory XKCD (`https://xkcd.com/327/`).

Stored Procedures

It is possible to define our own procedures in database server and save them and embed them into the database. Rather than having application logic stored solely in the application program we can embed some of it in the database. It can also enforce some separation of the logic: rather than the application program manipulate tables directly, the application calls the stored procedures and the procedures manipulate the database tables.

It is possible to write procedures in SQL (which is what we'll do in this course) but there may be support for writing it in another programming language (e.g., C or Java). One objective of this course is to be proficient in SQL so delegating everything to another programming language is undesirable. More than that, though, it is important to use procedures judiciously: simple SQL statements or even complex ones may be sufficient to get the job done; if everything is done in procedures then we miss out on the opportunity to learn to think about accomplishing the goal in the correct way.

It is worth noting that we can also define functions, which are similar to procedures but not identical. To reduce confusion and keep it simple we will focus just on procedures.

When we want to create a procedure, we need to give it a name, define the parameters and make a statement about whether the procedure is deterministic. Then there is the body of the procedure.

You will notice that nothing was said about return value. That's because a procedure does not have one: parameters are defined as `IN`, `OUT`, or `INOUT`. In parameters are the default and it means of course that it is an input parameter and changes are not made to the original input; out means it is a returned value so its initial value in the procedure is null until assigned; inout means it is taken as input and (possibly) modified before being returned.

On the subject of whether the procedure is deterministic, the mysql manual[8] says the following:

[8] `https://dev.mysql.com/doc/refman/5.7/en/create-procedure.html`

> A routine is considered "deterministic" if it always produces the same result for the same input parameters, and "not deterministic" otherwise. If neither `DETERMINISTIC` nor `NOT DETERMINISTIC` is given in the routine definition, the default is `NOT DETERMINISTIC`. To declare that a function is deterministic, you must specify `DETERMINISTIC` explicitly.
>
> Assessment of the nature of a routine is based on the "honesty" of the creator: MySQL does not check that a routine declared `DETERMINISTIC` is free of statements that produce nondeterministic results. However, misdeclaring a routine might affect results or affect performance. Declaring a nondeterministic routine as `DETERMINISTIC` might lead to unexpected results by causing the optimizer to make incorrect execution plan choices. Declaring a deterministic routine as `NONDETERMINISTIC` might diminish performance by causing available optimizations not to be used.

Enough introduction: let's look at a very simple example from [SKS11]:

```
CREATE PROCEDURE dept_count_proc( IN dept_name VARCHAR(20), OUT d_count INTEGER )
BEGIN
  SELECT COUNT(*) INTO d_count
  FROM instructor
  WHERE instructor.dept_name = dept_count_proc.dept_name
END
```

To create a procedure, the keywords are unsurprisingly `CREATE PROCEDURE`, followed by the name of the procedure. We observe that there is one in parameter and one out parameter. The order does not have to be in parameters before out parameters. The body of the function is bracketed with the `BEGIN` and `END` statements that are like the opening and closing braces on a function in a C-like language.

In a practical sense we probably want to have multiple statements and when we are giving in the create procedure statement we do not want a semicolon to result in detecting the end of the statement too early. The common solution to this is to bracket the create procedure statement with statements that change the delimiter (end of statement code). The syntax is `DELIMITER //` which then changes, from that point on, the delimiter to be `//`. Then at the end, we change it back to a semicolon with `DELIMITER ;`. The double slash delimiter is just convention, it could be anything, and some texts use `$$` instead.

```
DELIMTER //
CREATE PROCEDURE proc ()
DETERMINISTIC
BEGIN
  DECLARE a INT;
  SET a = 42;
  INSERT INTO table1 ( a );
END//
DELIMITER ;
```

The procedure above shows the use of the delimiter changing syntax as well as putting DETERMINISTIC and a multi line procedure, and some new things we haven't seen before such as declaring variables and assigning them. To provide a quick summary of the syntax from [Lev11] we can do the following things: declare variables, assign them, and use flow control structures like if-statements, and we can iterate with a cursor.

Declaration of a variable is pretty simple, you simply declare the variable using `DECLARE` with a name and data type (and an optional default value) such as the statement above. To assign it, use `SET` as above.

To understand how if-statements work, we'll do a comparison against a typical C like language:

```
if ( x == 0 ) {
  y = 1;
}
```

```
IF x = 0 THEN
  SET y = 1;
END IF;
```

Aside from the fact that the if condition is no longer in brackets and the equality is tested with a single equals sign, the big difference is the presence of the `THEN` as well as `END IF` keywords, in place of the usual curly braces. We can extend that with an else block as well:

```
if ( x == 0 ) {
  y = 1;
} else {
  y = 2;
}
```

```
IF x = 0 THEN
  SET y = 1;
ELSE
  SET y = 2;
END IF;
```

There are of course loops (and there's a goto statement, but please don't...). We'll cover the while-loop, but it's not the only kind there could be; there are also other keywords...

```
while ( y > 0 ) {
  y = y - 1;
}
```

```
WHILE y > 0 DO
  SET y = y - 1;
END WHILE;
```

If you want to iterate over rows returned by a query and perform some action on them, then the command for that is `CURSOR`. Use of a cursor is complex and there are several parts we need to do.

```
DECLARE e VARCHAR(128);
DECLARE quit BOOLEAN;

DECLARE cursor1 CURSOR FOR
  SELECT email FROM USERS;
DECLARE CONTINUE HANDLER FOR NOT FOUND
  SET quit = TRUE;
OPEN cursor1;

loopname: LOOP
  IF quit = TRUE THEN
    LEAVE loopname;
  END IF;

  FETCH cursor1 INTO e;
  # do something useful with e

END LOOP loopname;
CLOSE c1;
```

Creation of a cursor is done with the `DECLARE` statement as shown; it takes the result of a select statement. That is the set that we will iterate over. Then we need to indicate our ending condition, what to do when we get to the end of the data set. The usual correct answer for that is to break out of the loop and go on to the next step, as is done here. Then the `OPEN` command actually carries out the select statement and loads the data for the cursor to iterate over. Then we have our loop; we use the `FETCH` command to load the next variable into e; after which we can do something useful with it (not shown here for space reasons). It's worth noting that the loop has a name (loopname) which we use to break out of the loop if the quit condition is fulfilled. After the loop there is a close statement which frees up the memory allocated for the cursor.

Cursors don't update their data set if the underlying table is changed in the meantime; that's why the open statement exists and why its placement can matter. Open it at the last minute to get the most up to date data. Cursors are read only, and they always advance from one item to the next and never go backwards or skip anything.

By default, the procedure you create has autocommit set to "on" so any changes made are performed immediately, even if you have a multi line statement. That is not necessarily what you want; it may be that you want the whole block of statements to be processed as a single transaction (yes!). If that is the case then some additional syntax should be added. At the beginning of the transaction, add `START TRANSACTION;`. If everything goes well and you are ready for your changes to be saved, use the statement `COMMIT;`. If for some reason you need to cancel your

changes and don't want them to be saved, then use `ROLLBACK;` instead of the commit statement. As fun as it might be to go through a detailed example, that might give away a little too much for one of the labs...

To trash a procedure, the syntax is just `DROP PROCEDURE dept_count_proc;`. There does exist a limited ability to alter a procedure, but don't do it; if you need to change a procedure, it is best to drop and re-create it.

Finally, to call a stored procedure, the keyword is `CALL` followed by the procedure name, and, obviously, the arguments the procedure needs in parenthesis. If there are no arguments needed, empty parenthesis are used.

Triggers

It is possible to define an operation in the database that will take place automatically when some other modification of the database takes place. As you might imagine from the name, it follows a logic of "if x happens, then do y".

To define a trigger we need to define [SKS11]:

- An event that causes the trigger to be checked.
- A condition that must be satisfied for actions to be taken.
- The actions to be taken.

Triggers are useful in a few scenarios. They can be used to cause some events to occur such as updating related data, checking an integrity constraint that would be too difficult to check any other way, or alerting humans that some condition is satisfied. I generally discourage the first use: application logic should probably be the one that modifies the data. Checking integrity is a good use. Alerting humans is usually something like sending an email... No program is complete unless it sends e-mail.

Let's do an example from [Buc11] where there is a "blog" table with blog posts (sure, why not) and an audit table that stores deleted entries so they can be recovered if we want.

```
DELIMITER $$
CREATE TRIGGER blog_after_insert
  AFTER INSERT
  ON blog
  FOR EACH ROW
  BEGIN

  IF NEW.deleted THEN
    SET @changetype = 'DELETE';
  ELSE
    SET @changetype = 'NEW';
  END IF;
  INSERT INTO audit (blog_id, changetype) VALUES (NEW.id, @changetype);

  END$$
DELIMITER ;
```

There are further options, of course, and this is only a specific example. The `AFTER INSERT` statement tells us the time (either before or after) and the event is one of `INSERT`, `UPDATE`, or `DELETE`. If there are multiple triggers on the same condition we can specify an order on them by name, saying that it `PRECEDES` or `FOLLOWS` some other trigger. And then we have the trigger body which starts with the for-each row and begin statements. The operations of the body are done on each row that satisfies the conditions of the trigger

In the body, the row being inserted is pointed to by the `NEW` keyword. In an update there is also the `OLD` keyword to reference the old row. A delete has only the `OLD` row and there is no `NEW`.

The example does not have a `WHEN` condition, but it is possible to have one which allows us to specify a condition that limits when this trigger performs the action (without an if block where one branch is blank).

7 — Security

Security, Report to the Bridge

In a lot of books, and even in my original plan for how the lectures would be carried out, security was left to the end. This strikes me as being a disservice: security is something you want to bake into your product and something you want to have in your mind when you are designing it. It doesn't work to try to bolt it on afterwards.

Just as a matter of terminology: an *attacker* is a malicious user who is trying to damage or exploit the system. A *countermeasure* is some action or process we can take or implement to defend against attacks. An attacker is successful if he or she gains access to information he/she shouldn't have, damages the system in some way, or interferes with its legitimate operations.

In this lecture we will not go into the legal or ethical issues related to security, or security policies that are designed by organizations to tell you what keeps the system secure. We will focus on how access is controlled in the database, and specifically about the SQL Injection attack.

The truth is that databases contain a lot of data and some of it might be sensitive. In fact, there are several ways a piece of data could be declared sensitive. It might be that it is personal information, such as medical history of a patient or an employee's home address. It could also be that it comes from a sensitive source. The sensitive data might not be the entire tuple; it might just be a particular attribute such as Social Security Number or salary [EN11].

Regardless of why some data is sensitive, database administrators must see to it that the security policies are enforced. This means specifically that sensitive data is protected, but also that data is not corrupted and that access to any data is limited to those who should have access. Security policies do have some tradeoff with usability in that it can be frustrating for users who are denied some operation and must instead ask administrators to do it for them. But you also can't be too lax about this, because you most certainly do NOT want to find your company's name on TV having to report a data breach in which user personal data was stolen.

The database has a few different control measures to provide security, but for the most part we will talk about access control. That is, what users have access to what in the database. If this is configured correctly, then a lot of the security (although not all) is handled by the database server. But like seatbelts, access control doesn't work if you don't use it.

It is the job of the database administrator, mostly, to control accounts and privileges. This is analogous to the administrator or root or superuser account in operating systems, in that they have permissions to do everything and anything... but it is not good practice to log in using that account... It is better to create limited accounts and assign only the exact privileges needed to those accounts...

Access Control

I expect it comes as no surprise to you that database systems have accounts and user accounts have passwords. Users are expected to log in with their names and passwords when database access is needed. Whether they are connecting at the command line or using some sort of GUI client, the user information needs to be given in. The user name and password will be checked by the database server and if they match, the user is granted access. Otherwise, access denied.

The database itself will likely use some tables internally (in MySQL this is the "information schema"; it calls itself `information_schema`). When accounts and permissions are created they are recorded in the database's internal tables. After all, the data could be organized using the toolset we already have... relational database tables.

In addition to that, when a database user has successfully logged in, they have an associated *login session*, which is associated with all of the interactions that user does while logged in. This sort of information can be used to track who has logged in and who made what changes so that administrators can see the history in case something has gone wrong. If any malfeasance is suspected, database administrators can audit the logs (look through them) and determine what happened [EN11].

Privileges can be given out at the account level: these are per-account and includes the create schema, create table, create view, alter, drop, modify, select...

Users are created with syntax like: `CREATE USER 'exampleuser'@'server' IDENTIFIED BY 'weakpassword';` This creates a user called "exampleuser" at a particular database server ("server", often identified by an address) and their password is "weakpassword" (which is a terrible password, as you might know). To delete a user, `DROP USER exampleuser;`. To change a user's password, `SET PASSWORD FOR 'exampleuser'@'server' = PASSWORD ('CorrectHorseBatteryStaple');` The use of the `PASSWORD()` function is important, because it is what encrypts (well, one-way hashes, most likely) the user's password and that form is what is stored.

The next level is at relation level. Relations typically have an owner, usually the account used when the relation was created. It is likely that when a database schema is being created for the first time, an administrator account (eg root) is doing the creating. And the owner can allow other users to access it by *granting* them access [EN11].

Granting and Revoking Privileges. The keyword for this is `GRANT` and we'll soon see how it all works. The following privileges may be granted [EN11]:

- **Select** – Retrieve data or read data from a table. Namely, this allows the select statement to be used on R. The select statement can be as complex as desired, but may not change anything.
- **Modification** – Allow modification of tuples, notably through update, delete, and insert statements. The insert and update privileges may be limited to just certain attributes.
- **Reference Privilege** – The ability to reference a relation when specifying integrity constraints. This can also be restricted to certain attributes.

Obviously, if privileges can be granted, they should also be able to be revoked. For that purpose, of course, there is the `REVOKE` command.

Our basic syntax is as follows: **GRANT** <privilege list> ON <relation> TO <user list>; So if we want to allow selection for user Alice on the table for books, it is `GRANT SELECT ON books TO alice;` . We can be as specific as we want to be: `GRANT UPDATE books( price ) TO bob, charlie, donna;` – this allows the users Bob, Charlie, and Donna access to update the price of books but nothing else. If we didn't specify any particular attributes, it would be possible to update all attributes of the relation. We can also grant `ALL` to give all privileges.

It is possible to grant permissions to a special username, "public", which assigns the permission to all current and future users of the system [SKS11]. I really recommend against this sort of broad authorization. It goes against the idea of the principle of least privilege: user accounts should have only the privileges they need and nothing more.

The revocation basic syntax is as follows: **REVOKE** <privilege list> ON <relation> FROM <user list>; So if we no longer want to allow selection for user Alice on the table for books, it is `REVOKE SELECT ON books FROM alice;`.

If you're wondering why we might ever want to grant the reference privilege, consider the following scenario: the foreign key constraints restrict deletion and update operations on the referenced relation. If someone creates a table B referencing another A, then an attempt to delete an element from A may fail due to the foreign key constraint added by B. Thus, it is sensible to have the references privilege, since the ability to add a foreign key restricts future activity by other users [SKS11].

It is easy to just ignore all this and allow all your users to have all permissions, but this is dangerous. So, you say, maybe just DB administrators and developers. In particular, you might not want to allow developers this access either. Manual changes to the database should probably be done via some sort of double-check system where one developer writes an SQL script, gives it to another developer for review, and then a DB administrator carries it out. Without that, a hastily written SQL statement could easily wipe out lots of data... a forgotten where clause could set all voucher numbers in the accounting area to be the same value... or a delete statement might trash more than it was originally intended to. You might, however, find it helpful to allow developers read-only access so they can look at the database and potentially diagnose some problems.

Giving out permissions individually to users might be rather tedious; every time a new user is added it might be necessary to run hundreds of grant statements to that new user for each table. To manage the complexity, we might want to use *roles*.

Role-Based Access Control (RBAC). In Role-Based Access Control (RBAC), a set of roles is created, users are assigned roles, and access is granted or denied based on the role(s) a user has or lacks. Thus, assigning rights to users is done by assigning roles to users. A user can have more than one role, but must have at least one.

Example: only members of the accounting department may read the payroll information. Thus, the payroll relations are marked as being accessible only by accounting. When a user tries to access a payroll relation, the user's roles are checked. If the user's roles contains accounting, access is granted; otherwise access is denied.

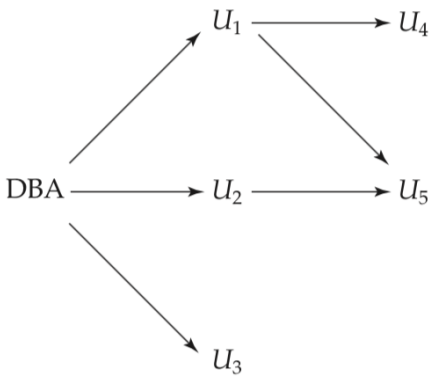
An advantage this has over direct management of permission granting is that assignment is simpler; if there are many relations which can only be accessed by the accounting department, it is much easier to assign a new employee the accountant role than it is to write a ton of grant statements.

Roles are created very simply in SQL: `CREATE ROLE accounting;` would create the role of accounting but it would not have, at this point, any permissions. Then, we can grant permissions to this role as if it was a user: `GRANT SELECT ON payroll TO accounting;` would do the job. But then you need to give some users the role: `GRANT accounting TO leslie;`. Now Leslie will have all the privileges that are granted to the accounting role, including ones added to the role after it has been granted to Leslie. In addition, Leslie will also have any roles granted directly.

There can also be relations between roles: doctors might be able to do all the things a nurse can do. Rather than have extra permissions assigned everywhere, the system can be set up so the doctor role *subsumes* the nurse role: the doctor role has all the rights of the nurse role, and may have others. To do that, grant one role to the other: `GRANT nurse TO doctor;`

Delegation. It is also possible to delegate – pass the authorization on to others – if the permission for that is also granted. In the examples we have seen so far, the privileges cannot be granted by the recipient to any other users. If, however, we added the `WITH GRANT OPTION` at the end of the `GRANT` command, it means that the recipient (e.g., Alice) may then grant that permission to someone else as well (e.g., Joe) by issuing a grant command herself.

The database server will want to keep track of what permissions have been granted to which users and by whom. We'll see in a moment why. But to do this, an authorization graph is maintained. A sample authorization graph is shown below:



A sample authorization graph [SKS11].

The graph can be checked for the existence of privileges; a user U has a privilege p if and only if there exists some path from the root (DBA account or root user) to U (however many steps that takes). This is probably inefficient though, as searching the tree on every transaction to see if the user doing the command has permission to do it is likely to take too long.

What the graph is actually for is revocation. If in the diagram, U_1 has the permission revoked by the DBA, the permission that U_1 has granted to U_4 and U_5 will be revoked as well. This, in a practical sense, has no impact on U_5 because the permission has also been granted to U_5 by U_2, but U_4 can no longer perform the operation.

You might think that you can trick the database: if U_8 grants the permission to U_9 and then U_9 grants it to U_8, if the database administrator revokes the permission grant to U_8, what happens? If we just used reference-counting, we would see that these users both have the permissions. But in constructing the graph we would see the problem immediately. In this case there is no path from the root to either U_8 or U_9, so authorization is revoked from both people [SKS11]. Keep in mind that there does not actually have to be a permanently maintained graph, but it might be necessary to construct it now and again to check the state.

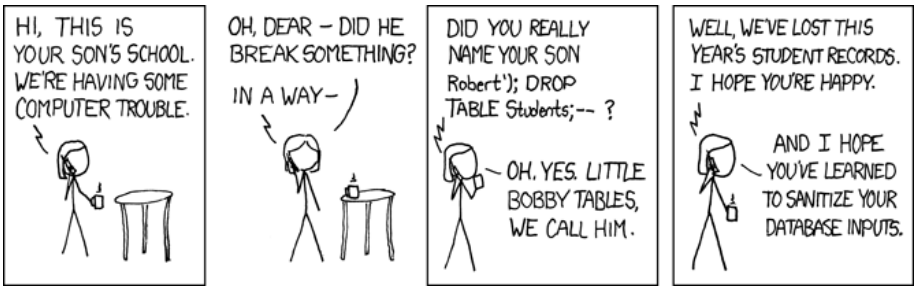
There is the possibility to modify the revoke command to prevent cascading revocation; in that case just put `RESTRICT` at the end of the statement. If that is the case, then if the permission has been granted to anyone else (i.e., it would cascade), the revocation will not happen and the system will return an error. The opposite of that is the `CASCADE` keyword, but it's unnecessary to use it because it is the default behaviour if neither restrict nor cascade is specified [SKS11].

To add one more wrinkle to this, sometimes we want to grant permission not from the current use but from a particular role. If that is the case, then add `GRANTED BY CURRENT_ROLE` to the grant statement.

Allowing users to grant roles is, in my opinion, fairly dangerous. Users are likely not super well equipped to see the consequences of granting permissions and it is likely better if permissions are centrally granted. That is, a database administrator is responsible for the whole process; if a user wishes to grant some permissions to someone else, that user goes to the DBA and asks that the permissions be granted. This sort of centralized management makes sure that there is an oversight process for who can do what and avoids unexpected behaviour when someone's privileges are revoked (e.g., an employee quits and it means members of that person's team can no longer access the complete data of the application).

SQL Injection

It turns out that you already know quite a lot about the basics of a SQL injection attack. Recall this XKCD from the previous topic:



Obligatory XKCD (`https://xkcd.com/327/`).

In short we get into this problem wherever data is entered into the database, usually through an application program. If there is a field for "name" we can expect that most users will try to put in their name, but some people will try to break the application. They will try to do this, as above, by putting ending the current statement and then executing an arbitrary statement after that. In the example above, after the first name there is `');` to end the previous statement, then the arbitrary command is to drop the table of students, and finally there is a `--` which comments out the rest of the line. And with that, the attacker is in, and your data is toast.

There are a few categories of things that can be accomplished by SQL injection [EN11]:

- **SQL Manipulation**: When you put in some SQL statement that adds or changes the conditions of the where-clause or otherwise does some set operation (union, intersect, etc). A common example is adding an `OR 'x' = 'x'` clause in a where which means the right side condition always returns true.
- **Code Injection**: See the example with little Bobby Tables above: introduce arbitrary SQL statement.
- **Function Call Injection**: Instead of performing a SQL manipulation with the injected command, call a system call or perform some other operating system function (or just make some invalid statements that crash the program... the DB server!).

And what can you do with this sort of access [EN11]?

- Execute arbitrary commands on the database (as we have seen).
- Determine what type of database is in the backend so we can see what is vulnerable.
- Denial of service attacks.
- Skip authentication (as above).
- Learn more about the internal structure of the application (mostly because error pages are very verbose).
- Privilege escalation – gaining access to things that you should not be able to access.

In general the solutions are pretty well understood: user input should be filtered (sanitized) and not allowed to be put directly into the queries. This alone might not be enough, and it is better to bind the user input to parameters rather than dynamically building query strings. In this course we don't talk about how to write an application program that interacts with the database (we just tell the database what we want). But do keep this in mind for the future.

Encryption

As another small note, data encryption is another way to protect your data. Certain sensitive data, such as passwords, must always be kept encrypted in the database. It should be encrypted with a one way hash function. We'll assume that you have some idea about how this works, but basically, if you don't: a security breach means that the passwords of your users will be leaked to the internet. And because users re-use passwords between sites, if someone's password on your service is leaked, it could very easily be someone's online banking password...

With that in mind, encryption keys should also not be stored in the database either... That would make it too easy for attackers! Or, if they are stored in the database, they can be stored encrypted. The standard sort of solution to this is to encrypt the key with the user's password. So the user logs in, the password decrypts the encryption key, and then the encryption key is used to decrypt the sensitive data. This might seem like overkill, but again, you don't want to see your company named and shamed on cable TV as having leaked the personal data of your users, do you?

8 — Modelling Diagrams

Modelling Diagrams

We have talked at some length about the tools for defining the database, but we really should spend significant time on good design. Knowing how hammers work is not enough; we also need to understand what good architectural plans look like.

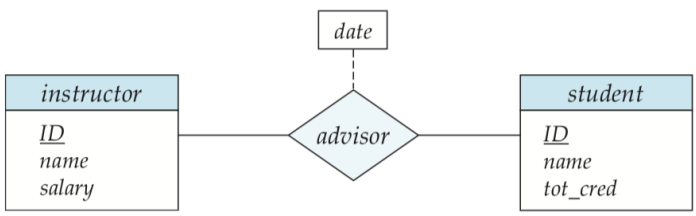
The first thing we would like to watch out for is redundancy in the database and eliminate it wherever possible. In short, if there is redundant data then there is the potential for data to get out of date or be in some other inconsistent state.

When we have some data that we would like to represent we need to turn it into some database tables. Going directly from some ideas about what data we need to the tables is sometimes difficult and may produce undesirable results. Instead, we should devise a plan for how it should be implemented and for that we will create diagrams. Diagrams are done on paper or a whiteboard or similar and it is very easy to visualize, understand, and change the design at that stage.

We are going to use Entity-Relationship diagrams (E-R diagrams) as our method for modelling the entities in the database as well as their relations. Entity-relationship diagrams have the following symbols and their meanings [SKS11]:

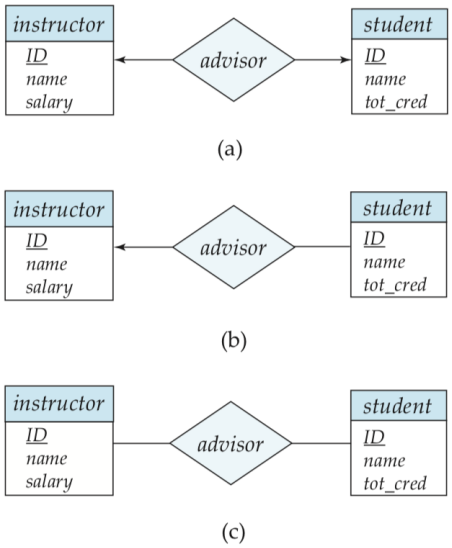
- **Divided Rectangles** represent an entity; an entity has a title (the top part) and a list of the attributes of that entity (the bottom part).
- **Diamonds** represent a relationship set.
- **Undivided Rectangles** represent the attributes of a relationship set.
- **Lines** link entity sets to relationship sets.
- **Dashed Lines** link the attributes of a relationship set to its relationship set.
- **Double lines** indicate total participation of an entity in a relationship set.
- **Double Diamonds** represent identifying relationship sets linked to weak entity sets.

We'll come back to those last two items later. Note also that in rectangles, attributes that are or form part of the primary key are underlined. Consider an example below:



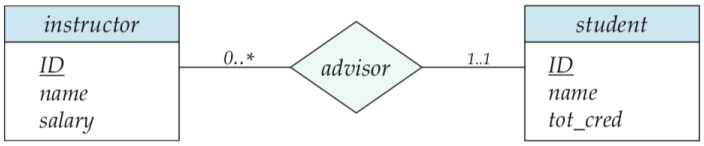
E-R diagram showing relationship between instructors and students, with attributes [SKS11].

In this case we have a relationship between instructor and student called “advisor” and that relationship has an associated date. What sort of relationship is shown here? This is a many-to-many relationship. To decide as to whether that makes sense, let’s think about real life: a professor can advise multiple graduate students, and a graduate student can have more than one advisor (for example, two of them). The diagrams below show the other possibilities:



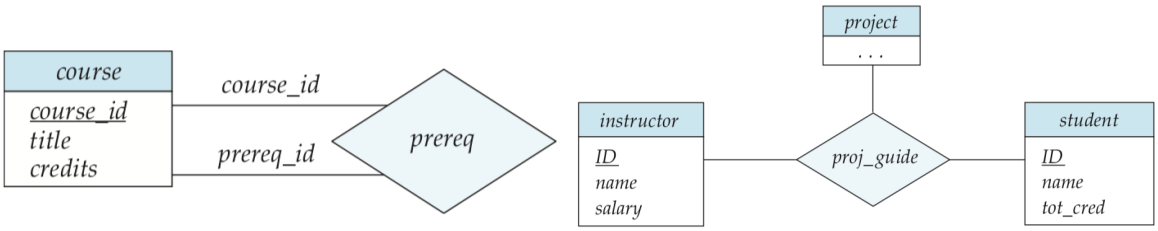
E-R diagram with (a) one-to-one, (b) one-to-many, and (c) many-to-many relationships [SKS11].

If we want to show some more specific rules that allow us to be more specific than just one vs. many, we can apply some numbers to the lines. In the diagram below, an instructor can have zero or more students and a student can have exactly one advisor.



E-R diagram showing cardinality limits [SKS11].

There is no reason why a relationship must be binary. A table may reference itself, as in the case of courses that have prerequisites. If that is so, then assigning names to the relations help to differentiate. Or a relation may involve many relations as in a ternary relationship:



Non-binary E-R diagrams; unary and ternary respectively [SKS11].

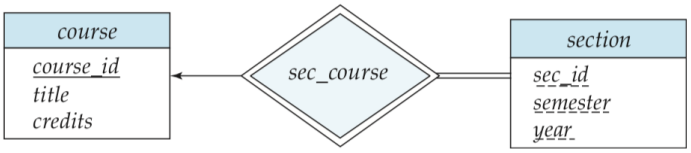
Weak Entity Sets. Weak entity sets are not entities that do not even lift. Instead, they are entities that are dependent on another entity, namely, one that cannot be identified uniquely based on its own attributes. The example that builds on this “university” model is a section: a section of a course, such as 001, is dependent on a course in a particular term.

Suppose that a course has an id, name, and code. A section has a section number, year, and term. A particular course, ECE 356, will have one or more sections, e.g. 001. But the attributes of the section on their own (001,

2018, Winter) are not enough to uniquely identify a single course. Because at least one another course, such as ECE 459, also has a section that matches (001, 2018, Winter). To differentiate between two sections that match on those same criteria we need a course number, but the course number isn't a part of the section entity. That is part of the course entity. And this is what makes the section entity a weak entity.

More formally, an entity that on its own does not possess sufficient attributes to form a primary key is a weak entity; one that is not weak is a strong entity. A weak entity set is associated with another entity, called its identifying set or its owner entity set [SKS11].

In a practical sense the weak entity will probably have some attribute added to it that identifies which strong entity the weak entity belongs to. That is called the *discriminator*. That would mean some attribute added to the section entity that references a particular course. Still, that is an implementation detail that plays no role in the E-R diagram, which looks like the diagram below:



E-R diagram with weak entity set [SKS11].

It is worth noting that the weak entity shown in the diagram has some slight differences versus a strong entity. The attributes are underlined with a dashed line and the relationship is modelled with a double diamond. It also uses double lines between section and sec_course that indicates total participation: every section MUST be associated with a single course.

We could choose a different option, of course, and add some sort of unique identifier to the section that would promote the weak entity to a strong one. But that doesn't necessarily make logical sense in the context of the application; a section doesn't make sense on its own and giving it a unique identifier does not really correspond with reality: a section is logically dependent on the course, isn't it?

Weak entity sets can exist in other configurations than just the identifying relationship. A weak entity can be the owner of another weak entity, or a weak entity can belong to more than one identifying set.

If you'd like to think about when it would make sense to have a weak entity, you might ask yourself if a particular entity can exist independently of any other entity. In an e-commerce scenario, there are customers, who have orders composed of items. Customers can exist if they don't have any orders, and items can exist if they don't appear in any orders. In that case, we would expect that both customer and order both to be strong entities.

Continuing this analogy, what about product reviews? A review belongs to an item, and if it is not anonymous then it also belongs to a user. But a review does not make sense independent of an item, otherwise, what would it be a review of? In this case, review can be modelled as a weak entity and it can have one identifying set (item) or two (item and customer). If we wish, though, a review can be a standalone element with its own unique identifier (e.g., a review number so that it can be accessed by referring to review #1337 or similar), in which case we would promote it to a strong entity.

Of course, the database can't be composed entirely of weak entities; there have to be at least some strong entities.

Extended Features of the E-R Model

The basic E-R model that we have discussed up to this point covers almost all of what we need to model a database, but there are some extensions that would be nice for notational convenience or ease of understanding.

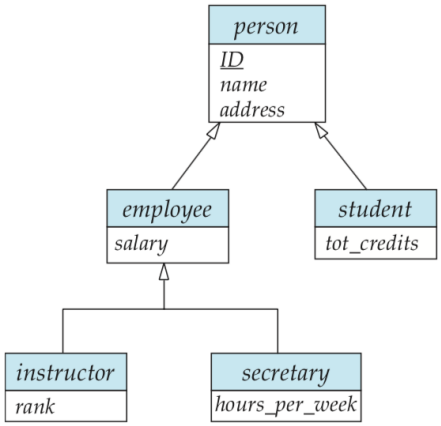
Specialization & Generalization. The first idea that we will look at is specialization. This is the same as a subclass in object-oriented-programming. We have some parent class (e.g., user) that has some number of attributes like ID, name, email, et cetera. And then there are some particular classifications: a staff member is a user, but

has certain staff specific attributes like office and phone number. A customer is a user but has different attributes like frequent flyer number, airline status, et cetera.

The mirror image operation of this is generalization, which is analogous to extracting a superclass in object-oriented programming. In that case we would identify some common attributes between certain entities and decide to consolidate the common attributes in a single entity from which the other ones are derived.

Obviously, this only works if we have attribute inheritance: sub-entities receive the attributes of their super-entities, in the same way that subclasses receive the attributes of their superclasses. As you are familiar with the ideas of object oriented programming (... right?) we don't have to actually go into any detail on that subject.

In OOP there are notations about declaring a property as private, protected, package-private, or public; these are irrelevant in the case of an E-R diagram. Actually, in a language like Java if an attribute is private it isn't accessible in a subclass. Those sorts of information hiding (encapsulation) principles aren't present in the database design.



An E-R diagram showing specialization and generalization [SKS11].

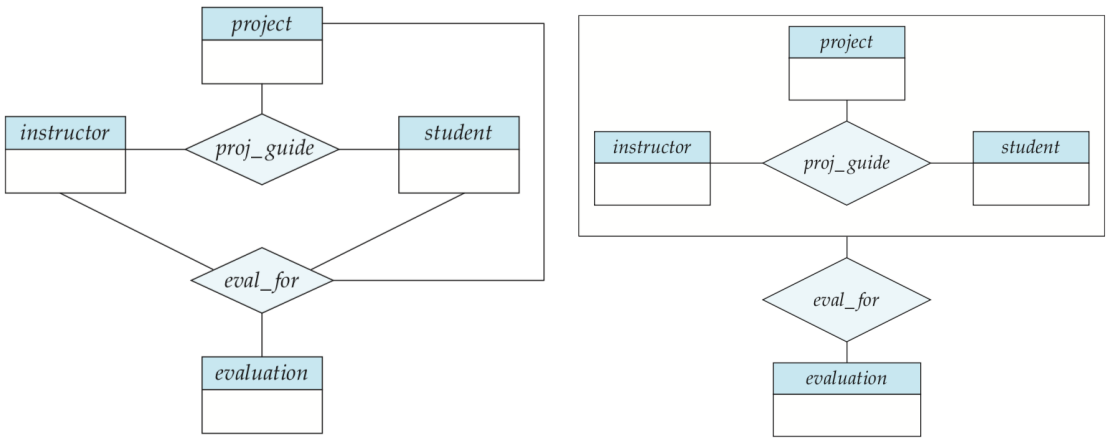
We have some choices about how the generalization is defined. The first option is to have it be *condition* defined, such as having some rules that say that a person who fulfills a certain condition is included in that group. So a customer who has status of "Frequent Flyer" in the system is automatically included in that specialization. In other cases, an operation must be taken to specifically include a person in that subset, such as assigning an employee to a specific department [SKS11].

We could also have rules that say sets must be disjoint (a member of some group cannot be a member of another group), or if they can be overlapping. In our system we could say that a staff person can also be a customer, but that is specific to the domain we are discussing.

We can also have the idea of an "abstract class": we could write in some rules that say that no instance of "user" can exist, and everyone must be a staff member or customer (or both). In that case, this is called total specialization: we are not allowed to have unspecialized (general) entities. The entity user might appear in the E-R diagram but as notational convenience only.

Aggregation. Aggregation allows us to treat a grouping of entities that are related as a single block. This can simplify the diagram and show us a relationship between relationships. Aggregation may not seem very useful in the context of a database schema that can be handily represented on a piece of paper, but for a large database it can be a good way to sequester parts of the design in a different area so the diagram is comprehensible.

The scenario we'll do has to do with a four-way relationship between instructor, student, project, and evaluation. In this scenario, hopefully a realistic one, there is a student project that is supervised and evaluated by an instructor. The evaluation belongs to the relationship "project guide", which is defined by the three elements of (instructor, student, project). Actually, the name project guide is probably unclear. If it were up to me I would call the "project" relation "topic" and the "proj_guide" relationship would be "project".



E-R diagram with complex relationship without (left) and with (right) aggregation [SKS11].

It looks like the relationship *eval_for* and *proj_guide* can be combined because they seem to connect the same three things. But we may not want to do that if some sets of (instructor, student, project) do not have an associated evaluation. Now possibly the data is redundant; if the evaluation is a simple enough element (e.g., it is an enumeration or an integer letter grade) then it could be an attribute of the *proj_guide* relationship. But this option is not suitable if the evaluation relation is used in some other relationship or in some other context. We also might not be able to combine it if there are multiple evaluations on a project... [SKS11].

Conceptually we might prefer to think of *proj_guide* as a higher level entity, leaving a binary relationship between that aggregated entity and the evaluation relation.

It is worth noting that the textbook contains a couple of alternative diagram notations. You may encounter them in the real world, but we will not invest any time discussing them as they could be confusing.

Given a good understanding of how diagrams are formed, we will next think about how to turn models into tables, and what makes up a good design.

9 — Reduction to Relational Schema, Design Decisions

Modelling Diagrams into Tables

If we have an entity-relation diagram, eventually we will want to turn that into a set of database tables. The conversion routine is not especially complicated and after a small amount of practice it is likely that you will be able to do it quickly and efficiently.

Strong Entity. Strong entities are the easiest to turn into tables. The name of the table will be the name of the entity (surprise) and the attributes in the diagram become the attributes in the relation. The primary key in the table will be the same as the primary key in the diagram. If the diagram says an instructor has *(id, name, salary)* and *id* is the primary key, then the create table statement will assign types and lengths to these fields.

Weak Entity. Weak entities are more interesting. Suppose A is a weak entity set and the strong attribute it depends on is B. The weak entity table is created using the attributes of A as well as the primary key of B, the entity it depends upon. If the weak entity is the section of a course, as before, and course is the strong entity, then the section entity would be created with all attributes of the section (weak) entity and the primary key of the course entity (course id).

It is also sensible to add constraints to this table that says the attribute(s) referencing the primary key of the strong entity must exist in the strong entity table (not null and foreign key). In the weak entity table, the primary key is formed by the key referencing the strong entity plus the discriminator [SKS11].

Suppose that the weak entity is a list of something and the strong entity a container of some sort. Suppose you have a shipment: a single shipment is made up of items and those items are, let's say, weak entities in the example. Then the primary key would be the unique identifier of the shipment plus the discriminator, where the discriminator is an integer identifying the position in the list. So if the unique ID for a shipment is ABC12345, then the items for that shipment would have primary keys be *(ABC12345, 0), (ABC12345, 1), (ABC12345, 2), (ABC12345, 3)*.... This also provides a nice way to sort them.

Relationships. A relationship is possibly represented as a table in the database as well. As a first step, we will define the attributes for that table as if it will be a standalone table. Later, we might combine the table with another one to eliminate redundancy or simplify (see below) but for now the first step will be the naive table creation.

The table is created using (1) the primary key of each of the entities participating in the relationship, plus (2) any attributes that have been assigned to the relationship itself. That's pretty much it; the real question is what should form the primary key for this table? It depends on the nature of the relationship [SKS11]:

- For binary many-to-many relationships, the union of the primary keys of each entity participating in the set forms the primary key.
- For binary one-to-one relationships, the primary key of either set (but no need for both); either is fine.
- For binary many-to-one (or one-to-many) relationships, the primary key of the entity on the "many" side is the primary key for the relationship.

- For n-ary relationships where none of the relationships are "to one", the union of the primary key from participating entity sets makes up the primary key for the relationship.
- For n-ary relationships where there is a "to one" participant, the primary key is the union of primary keys of all the OTHER participating entities (but not the one that's "to one").

The foreign key constraints should be added from the relationship table referencing the entities participating in the relationship. Now, some of the tables generated by this process are redundant and some can be combined to make things simpler and a little bit clearer...

Redundancy. Take a look at a relationship linking a weak entity to the corresponding strong entity. Weak entities are modelled as being many to one and the relationships have no descriptive attributes. Because the weak entity has the primary key of the related strong entity as one of its attributes (thanks the the relations being resolved into the entities it links). Thus, for this reason, a relation that specifically links the weak entity to the strong entity is fully redundant.

Example: the weak entity of a section of a course linked to a strong entity of a course [SKS11]. A section a section id, semester, and a year. It is related to a course that has a course ID. So when we have done the work of turning the relationship into some tables we'll find that the table for section we already have course ID as one of the attributes. So we would not need a table that actually links them anymore, as it would be totally redundant.

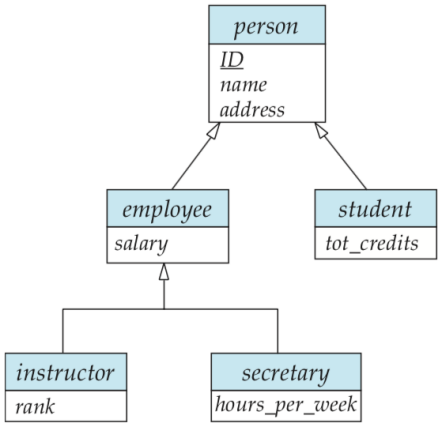
Combination. Sometimes our diagram presents us a number of entities which we would like to combine when we are creating tables (and we'll see in an upcoming lecture whether this is a good idea or not).

If a relationship exists between two tables A and B that is many-to-one (many A correspond to one B), we might naively create tables for A, B, and one for the relation AB. If it is (almost) always the case that A participates in the relation then we could actually combine A and AB to make a single relation that is the union of both attributes [SKS11]. Or we might decide that means that A simply gets an attribute that references B. If participation is not total, the use of null is appropriate.

The one-to-many relationship is just the reverse of the previous paragraph – instead of adding the attribute(s) to A, add the attribute(s) to B instead. In the case of a one-to-one relationship between C and D, the attributes of the relation can be added to either one of the entities, C or D, but not both. If we have a many-to-many relationship, then we cannot combine the tables (obviously).

If we have combined two tables, then obviously, we don't need foreign keys to link them anymore (because, of course, it would make no sense). However, a foreign key that would have appeared on a relationship AB that references some other entity needs to be created on the merged tables as well.

Generalization (Specialization). To transform a generalization into an entity set, we have two choices. The most obvious approach is something along the lines of what Java will do in the background when you create a subclass of an object: the properties of the superclass appear on the subclass. Recall this diagram from earlier:



An E-R diagram showing specialization and generalization [SKS11].

In this case then we would define tables for instructor, secretary, and student. We might or might not have one for employee, depending on our rules. If, of course, we don't allow "instantiation" of a person, but instead a person must be either an employee or student, we would not create a table for the person relation. You may liken this decision to whether the class person is abstract or not. So let's assume that instructor then created as *(ID, name, address, salary, rank)* with *ID* as the primary key. There will be some repetition here as the secretary relation will also have most of the attributes the same, except instead of rank it will have hours per week.

The alternative representation is to break it up so that we have instructor created as *(ID, rank)*, which necessitates creating employee as *(ID, salary)*, and person as *(ID, name, address)*. This does require a fair amount of repetition in the primary key attribute since it will appear in all tables. The primary key is then the only one that is repeated. With some foreign keys set up to make the IDs match up, of course.

That's interesting, both of these things will have some redundancy. But how much will we have? Depends: if the generalization is both disjoint and complete, the first approach is better. Disjoint and complete means that no entity is a member of two lower-level entity sets directly below a higher-level set, and if every entity in the higher level set is in one of the lower-level sets [SKS11]. If that's not the case, we might get some duplicate data, such as if a student can also be an employee: things like name and address appear twice for a person who is both. So perhaps the alternative approach is preferable if these two conditions do not hold.

Aggregation. Aggregation is actually relatively easy to work out. The tables for the elements inside the aggregation are created as normal. Then the rules for creating the key constraints can just as easily be applied to any relationship that involves the aggregation. The primary key of the aggregation is the primary key of its defining set, so just use that (and no relation is created to specifically represent the aggregation) [SKS11].

Design Decisions

In creating our E-R diagrams we have implicitly or explicitly made some key design decisions about how we would like to implement the design. In some cases there are better choices and worse choices, but in other cases there are alternatives that are equally valid that we could choose between.

Entity vs. Attribute. A decision we need to make frequently is whether a new piece of data should be added as an attribute or as a new entity. Let's say we want to add an e-mail address to users.

If the relationship is anything other than 1:1 then it is very unlikely we will choose a new attribute. How many e-mail addresses are users allowed to have in the system? If it is just one, our decision is more interesting. If a user can have multiple e-mail addresses, we must choose a new entity to represent e-mails. What about letting multiple users share an e-mail address? We might not want to forbid that specifically (although we can), but it still takes us back to the decision of whether a user can have one e-mail address or many.

Let's assume that a user can have exactly one e-mail address. Then putting it in an attribute is a viable choice. What if we put it in another entity? That might not be the best choice here, because we would have an entity for e-mail address that contains nothing but the e-mail address and then the attribute to link it to the user.

What if instead, we were thinking of adding multiple related fields instead of one field? Then it might make more sense to group those things separately in an entity of their own. An address, for example, is a number of related fields, street, city, postal code, et cetera. If a user has one address we could put all those fields on the user entity, but it might make more sense to move them to a different entity. Performance considerations may come into play (eventually) because large entities are unwieldy.

But the real decision is more philosophical... What is an attribute and what is better as a set? There is not a bright line between the two and it may depend more on the real-life situation being modelled than any actual technical consideration.

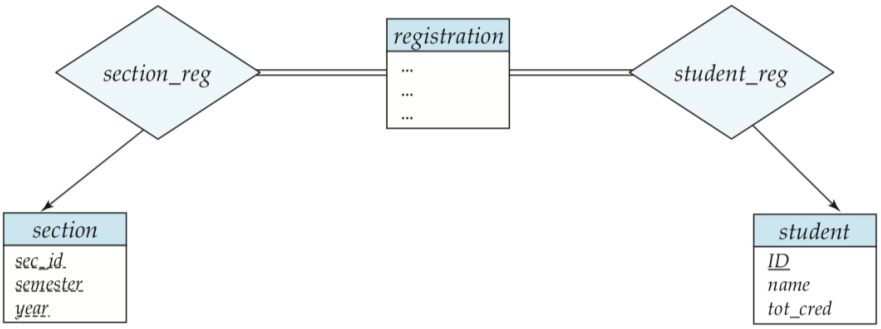
There is a wrong choice, though, and that is using the primary key of an entity as an attribute of another, instead of a relationship [SKS11]. Remember that in a relationship, in reality of course, the table will need to contain some reference to the other. But in the E-R diagram it should be represented as a relationship (especially because it allows the opportunity to assign more attributes to the relationship, that otherwise might not have been in the right place). Simply writing, for example, *student_id* as an attribute in a *instructor* entity as an alternative way of showing that there is a relationship (advisor) is suboptimal.

It's also worth noting that a common mistake is to write redundant attributes on the relationship [SKS11]. If a relationship is drawn between tables r_1 and r_2 in the diagram, there is no need to put the primary key of either r_1 or r_2 as attributes on that relation, because they are already implied by the fact that there exists a relationship between the two types.

Entity vs. Relationship. We are also sometimes faced with the decision about whether to make a particular object an entity or a relationship. There are things that are clearly an entity (e.g., a customer) and those that are clearly a relationship (e.g., a shopping cart) but there are other things that fall within a marginal category where it could go either way. In an online shopping scenario, though, what about order history? This is different from the idea of a shopping cart because if an item changes attributes before purchase, the data in the cart needs to be updated... if it changes after, for example, the price decreases, it does not change the already-purchased orders.

There are arguments for the relationship approach: it eliminates duplicate data and ensures that if a product is updated then the latest data is shown in the order. That might, actually, be an argument for the alternative. If a product is purchased then it might make sense to "freeze" in place some of its data at the time of purchase (e.g., price) so it is perhaps advisable to make an entity for a purchase order made.

Another example in the university schema is what to do with registration: when a student takes a class should that be represented by a relation that says the student with the id x registered in the section y? Consider the E-R diagram as below:

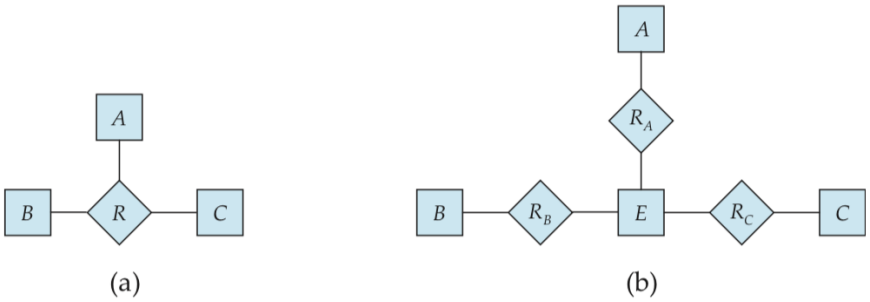


Registration entity created to replace a more complex relationship [SKS11].

My personal take on this is that this is worse than just representing student enrolment in a section as a simple relationship. This approach has more entities, more relationships, and is perhaps not as easily matched to a theoretical model of the university. That a student is registered in a course is less tangible than a shopping cart in the online shopping scenario, so it seems strange to make registration an entity. Then again, you could argue that registration corresponds to a card kept in a file somewhere but that is probably not how it really works (but the mysteries of the registrar's office are not for humans to know...).

Binary vs. Non-Binary Relationships. When we have a n-ary relationship, that is an n-way relationship such as we have seen before, we could choose instead to represent it as a number of binary relationships. Consider a work term report: it has an author (a student) and an evaluator (lab instructor). We could make it a three-way relationship or we could instead break it up into two relationships, one between work report and student, and one between work report and lab staff. If it was in one relationship, a work report that has no marker assigned yet would just have `null` as its assigned value until a marker is assigned.

It is always possible to replace an n-ary relation with some number of binary relationships, although we may need to put an entity in the middle of it to make it all work out. Suppose we have a ternary (3 entity) relationship R that connects entities A, B, and C. Any attributes of the ternary relationship become an entity E and then three binary relations are created: R_A relating A and E, R_B relating B and E, and R_C relating C and E [SKS11].



Converting a ternary relationship (a) into three binary relationships (b) [SKS11].

Even though it is possible to turn a ternary relationship into binary relationships, there are reasons why this is undesirable. In fact, we can immediately consider three reasons why this is not always a great idea [SKS11]:

- When a single relationship is split into two or more, it may no longer be clear the split relationships have any connection.
- An identifying attribute needs to be created for the new entity (E in the above example) and this needs to be stored and maintained. Then there would be more relationships which would make the diagram more difficult to follow and ultimately also mean more tables that are created in the database.
- Constraints that exist in a ternary (or more) relationship may be difficult or impossible to express in binary relationships. If a constraint says the relationship is many-to-one from (A, B) to C, i.e., each AB pair is associated with at most one C entity, this cannot be expressed with cardinality constraints on relations R_A, R_B, R_C.

Placement of Relationship Attributes. In the case of one-to-one and one-to-many relationships we could choose to create a table for the relation, but we don't have to... We could instead associate the attributes of the relationship with one of the entities. If we have a work report entity and we want to relate it to its author, we could have a table that relates the two... or a work report entity might end up with an attribute for the student ID to connect the two. But this wouldn't work for a book and author relationship, because a book can have more than one author and an author can write more than one book.

This discussion takes us back full circle to the entity vs. relationship discussion from earlier. If the relationship to be modelled is a many-to-many relationship then there isn't really a lot of choice. If an attribute A in the relationship is determined uniquely only by the combination of the entities' identifiers, then A needs to be placed on that relationship and cannot be put on either side. If the relationship is a shopping cart in an online store, and a discount code has been applied, the discount code's application belongs neither to the customer nor any individual item in the cart.

Security. Remember, permissions are handed out at the level of the table and that makes it sometimes necessary to divide some tables so that the appropriate security constraints can be applied.

Conclusion. There are, fortunately, some formal and precise ways for making a determination about whether our designs are appropriate. We would like our designs to be "normal" and there is a precise definition of what normal is.

10 — Normalization

Normalization

So far, we learned the syntax to create tables, and how to turn modelling diagrams into tables and all the various bits and pieces about how modelling diagrams work. The modelling diagrams give us some guidance and tell us about wrong ways to represent the data, but we don't yet have enough guidance as to what is correct. If we do a good job the relation will be in a *normal form*. Yes, *a* normal form... there are several.

An example from [SKS11] gives us an opportunity to do something wrong and show bad design: if we combine instructor (having the attributes id, name, department, salary) with department (having an id, faculty, budget) then our combined table means one larger relation replaces two smaller ones. This isn't total nonsense because instructors do have a relationship with a department, but is this a good way to model the data? Intuition might tell you no, but think about why. In the combined relation, if there are two instructors, say, Sedra and Smith, both members of department of electrical & computer engineering, in both tuples there will be an entry for the budget of the department. The budget data is duplicated and risks becoming inconsistent.

Our intuition may tell us this is bad, but we would like a way to express it formally. Informally the rule is that each value of department name corresponds to at most one budget. The formal description is a *functional dependency* and it is written $dept\_name \rightarrow budget$. A functional dependency specified that if there is a schema that consists just of the attributes for department name and budget, then the department name attribute could be the primary key [SKS11]. Combining instructor and department breaks this rule, because we have duplicate entries for department and therefore this can't work.

The functional dependency shows us that data is duplicated and that indicates that we need to split the combined relation in a process called *decomposition*.

The previous example is very egregiously and obviously wrong, making it simple to identify that there is a problem. In reality, a database will have many more tables and it will be harder to find out what the functional dependencies are, but there is an algorithm for that which will be covered soon enough.

Another example from [SKS11] shows us that decomposition on its own is not always good: we should split up things that don't belong together, but it is possible to go overboard. If the relation is *employee(ID, name, street, city, province, postalcode)*, we might think that we can decompose this into two schemas: *employee(ID, name), address(name, street, city, province, postcode)*. Does this work? No – two employees could have the same name and that is likely in the real world, as some names are extremely popular.

If instead of defining the address relation to be based on name, we could use the ID instead, and it would work: *employee(ID, name), address(employeeID, street, city, province, postcode)*. This is an acceptable decomposition and we call it *lossless* because no information is lost. The first attempt at this caused a loss of information (if two employees have the same name, there is the possibility of confusion), so it is called a *lossy* decomposition. We do not perform lossy decompositions.

Atomic Domains and the First Normal Form

Our E-R model allows attributes to be non-atomic, that is to say, divisible. If a course code is "ECE356", that attribute is non-atomic because it can be divided into two parts, the "ECE" (department) and "356" (course number)

components. The same is true of something like an address: "48 Main Street Southwest Unit 2A" might be the street attribute, but that is divisible into several parts. Those are non-atomic attributes values. If the domain is indivisible units, e.g., an integer, it is atomic.

A relation R is said to be in the *first normal form* (abbreviated as 1NF) if the domain of all attributes of R are atomic. If some attribute is not atomic, then we could subdivide it to put it into 1NF. Instead of course being "ECE356" we could split it up into department "ECE" and number "356". The primary key could still be composed of those two elements, mind you.

Does this mean that a string (varchar) attribute can never be atomic, as one may take a subset? No, what really matters is how they are used in the database. If the application logic requires breaking up the attribute for some reason, then it is non-atomic. The "ECE356" example is not atomic because there are situations where you would want to subdivide this. To answer the question "how many courses in the Winter 2018 term are ECE courses?", we need to look at all course codes and figure out which ones begin with "ECE". If employee numbers are generated in the format "AA1234" and can never be subdivided (e.g., the first two letters don't indicate anything like department) then they are atomic.

More than this, though, we don't like set-valued attributes. If we wanted to model all the departments that belong to a faculty, what we don't want is for the faculty tuple to have an attribute called departments which then contains multiple elements... For example, Engineering is a faculty, and the list of departments would be ECE, MME, SYDE, etc. This would not be in the first normal form, because the attribute of departments could be split up into each department. This can cause confusion and redundancy. What about Software Engineering? It belongs to the faculty of Engineering and Math... Data could be related.

Decomposition with Functional Dependencies

The schema that we create is supposed to model entities and their relationships in the real world. The real world understanding tells us important information about how the data should be modelled: students have a student ID number and the student ID number is unique. If the database does not represent that information in some way, we have done something wrong. The real world constraints may be represented as a key or as a functional dependency.

The schema is just the data model, and we care if the data in the tables conforms to the constraints. If the data in the table meets all the real-world constraints, that instance of the relation is called a *legal instance*. If all tables in the database are a legal instance of that relation, then we can say the instance of the database is a legal instance of the database schema [SKS11].

A *superkey* is a subset K of a relation R if, in any legal instance of R, for all pairs, of tuples t_1 and t_2 in the instance of r, if $t_1 \neq t_2$ then $t_1[K] \neq t_2[K]$ [SKS11]. This is to say that no two tuples in a legal instance of the relation may have the same values the subset of attributes K, or if you prefer, the attributes in K allow a tuple in the relation to be uniquely identified.

Suppose that attributes α and β are part of a relation r. A functional dependency $\alpha \rightarrow \beta$ is satisfied if for all pairs of tuples t_1 and t_2 such that $t_1[\alpha] = t_2[\alpha]$ it is the case that $t_1[\beta] = t_2[\beta]$. If this functional dependency holds on every legal instance of the relation, we can say the functional dependency holds on the schema [SKS11].

If it helps you to imagine this, a functional dependency tells us that for a given value of x, we know what the value of y will be. In this way it is like a function in math, if we know the value of x we know the value of $f(x)$ (namely, y). And most importantly, for every value of x there is at most one y.

We can say that K is a superkey of R if the functional dependency $K \rightarrow R$ holds on this instance of the relation. That means, formally, that if $t_1[K] = t_2[K]$ then it means $t_1 = t_2$ because a superkey identifies a tuple uniquely.

Functional dependencies are used for both design and verification purposes. We use them in the design process to identify and record the constraints that our database schema should be designed around. They will be used as a way to decide what entities should be formed and how their relationships to others appear. And when we have a set of functional dependencies, we can use them to verify if an instance of the database is legal; that is, check if all constraints are satisfied. If the answer is yes, then we say the set of functional dependencies holds.

A	B	C	D
a_1	b_1	c_1	d_1
a_1	b_2	c_1	d_2
a_2	b_2	c_2	d_2
a_2	b_3	c_2	d_3
a_3	b_3	c_2	d_4

A sample relation with dummy data [SKS11]

In the sample data above, we can identify the dependency $A \rightarrow C$ because for each tuple where the value of A is a_1, the value of C is the same (c_1), where A is a_2, C is c_2, and where A is a_3, C is c_2. That is, by knowing the value of A we can know the value of C. We can also see that $C \rightarrow A$ does not hold.

You are probably thinking that this relation has only five tuples in it and it may just be a coincidence that $A \rightarrow C$ holds on this relation. And that is entirely possible. Our data is observational here, not prescriptive. Without knowing a bit about what A and C are we don't know if it should be the case. Based on what we have here we can only verify if certain functional dependencies hold, if they are provided.

There are also *trivial* functional dependencies because they are always true and not even the smallest bit interesting. The functional dependency $A \rightarrow A$ is an example of a trivial dependency, and not very useful. A formal definition of a trivial relation is that $\alpha \rightarrow \beta$ is trivial if β is a contained within α (in set thinking) [SKS11].

Functional dependencies can be transitive: if $A \rightarrow B$ and $B \rightarrow C$, then $A \rightarrow C$. For each A there can be only one value of B and for that value of B there is only one corresponding C. If a set of functional dependencies is F, the notation that includes all of these implied dependencies is F^+, the *closure* of F [SKS11].

Boyce-Codd Normal Form

The *Boyce-Codd Normal Form*, known as BCNF, eliminates all redundancy that can be found using functional dependencies (but it does not eliminate all redundancy). A schema R is in BCNF with respect to a set of functional dependencies in F, if, for all functional dependencies in F^+ of the form $\alpha \rightarrow \beta$ either (1) the functional dependency is trivial, or (2) α is a superkey for schema R [SKS11]. A database is in BCNF if all its relations are in BCNF.

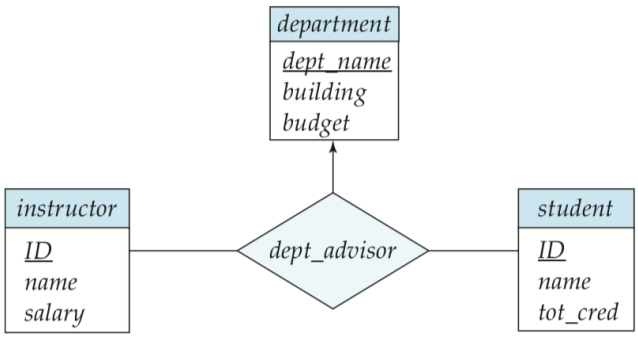
Calling back to the earlier example where we looked at merging instructor and relation, we can see that this is not in BCNF fairly easily. A department has a budget, yes, but in this relationship a the department name is not a superkey because we had two entries (Sedra, Smith) that were in the same department and those had the same budget. A department cannot have two budgets.

Our real-world understanding tells us that there is a dependency that says a department has a budget, and only one budget (even if the department chair would like to have the sum of the budgets...). That tells us there should exist a functional dependency. Given that a functional dependency should exist, check it against he conditions for BCNF. It is not trivial, nor is department name a superkey for this combined relation.

The general approach to decompose things that are not in BCNF from [SKS11]. If it is not in BCNF there is at least one nontrivial functional dependency $\alpha \rightarrow \beta$ where α is not a superkey for R. Then we need to split up R into two relations: (1) $(\alpha \cup \beta)$ and (2) $(R - (\beta - \alpha))$. The rule is stated in this way because sometimes we need to have attributes that appear on both sides of the arrow. We may need to repeat that process on one of the new relations until it is in BCNF.

Dependency Preservation. Avoiding duplication of data is not our only goal in a database design. Constraints also need to be checked. Although we have not yet seen how joins are performed, or why they are expensive (really), for now we should just understand that joins are expensive and we would like to avoid them. Checking if a constraint is satisfied may require a computationally expensive join if the relations are in BCNF.

Consider an example from [SKS11]: the previous relationship between student and advisor has been altered. Now, if a student wishes to have multiple advisors, that is permitted, but the student may only have at most one advisor from a given department. How realistic this scenario may be is not really the goal.



E-R diagram for ternary advisor-student-department relationship [SKS11]

The relation in the diagram above to represent *dept_advisor* will contain three attributes, all of which are primary keys in the other relations: *(student_id, instructor_id, dept_name)*. Based on what we know, the functional dependencies are *instructor_id* $\rightarrow$ *dept_name* (an instructor belongs to one department), and *student_id, dept_name* $\rightarrow$ *instructor_id* (a student has at most one advisor per department).

The usual definitions of instructor, student, and department relations are simple enough to imagine. But the interesting part is the relationship *dept_advisor*. Its definition would be: *(instructor_id, student_id, dept_name)*. What if we check this against the BCNF rules?

This design is not in BCNF, because *instructor_id* is not a superkey. An instructor ID alone is not enough to identify the relationship, because multiple students can be advised by the same individual. So we need to perform the decomposition procedure and that will produce two relations: *(student_id, instructor_id)* and *(instructor_id, dept_name)*.

Those are in BCNF, but it is difficult to verify the constraint that requires us to verify that each triple of *(student_id, instructor_id, dept_name*) is unique. That is because performing this check requires a join on the two relations in the previous paragraph. That is computationally expensive, as we will soon see.

Because it is now hard to check this dependency, the design is said to not be *dependency preserving*. It is not that it is impossible to check the dependency, although the terminology might seem to imply that data is somehow lost. It is not lost, just harder to check and verify.

Third Normal Form

The Third Normal Form, or 3NF, is weaker than BCNF that allows dependencies to be preserved. Now wait, you say, third normal form? What happened to the second normal form? Is BCNF the second normal form? No, it is separate from BCNF. A table is in the second normal form if it is in the first normal form and all non-key columns are dependent on the table's primary key [Wen14]. The second normal form is not especially popular and is mostly here for completeness. It is the third normal form that we are interested in.

The 3NF rules are a small modification of the BCNF rules. In particular, we would like to allow some nontrivial functional dependencies whose left side (the α) is not a superkey.

A schema R is in 3NF with respect to a set of functional dependencies in F, if, for all functional dependencies in F^+ of the form $\alpha \rightarrow \beta$ one of the following holds: (1) the functional dependency is trivial, or (2) α is a superkey for schema R [SKS11], or (3) Each attribute A in $\beta - \alpha$ is contained in a candidate key for R. Any one of those conditions suffices. Remember also that a candidate key is a minimal superkey. Note also that the third condition does not mean a single candidate key must contain all attributes in $\beta - \alpha$; as long as each attribute appears in at least one candidate key [SKS11].

Remember that a schema that is in BCNF would satisfy case 1 or 2 for each of the functional dependencies, so anything in BCNF is already in 3NF. 3NF, however, is slightly less restrictive and allows functional dependencies that are not permitted in BCNF, so something in 3NF may or may not be in BCNF.

Recall: the relation in the diagram in the previous subsection to represent *dept_advisor* contains three attributes, all of which are primary keys in the other relations: *(student_id, instructor_id, dept_name)*. Based on what we

know of the functional dependencies, this relation is already in 3NF. The step we took to do the decomposition to BCNF is unnecessary here if we can accept 3NF. There are tradeoffs, but we can come back to what those are...

How do we know the new definition of *dept_advisor* is in 3NF? We know that the current definition has a functional dependency that does not meet condition 1 or 2, otherwise it would be in BCNF. So the rule we will check is the third rule and we'll find that this is valid because *instructor_id* is contained in a candidate key: *(student_id, instructor_id)* is a candidate key, because it is minimal and sufficient to identify any row. If the student with ID x999chan is supervised by Professor Sedra, we have enough information to figure out that the department is ECE.

Higher Normal Forms

This isn't even my final form! In addition to the first, second, third, and BCNF normal forms there are more... Fourth, fifth, and beyond.

The fourth normal form is built upon the third normal form, so the first requirement for something being in 4NF is that it is in BCNF. The new requirement is that a record type cannot have more than one multivalued dependency. A multivalued dependency exists between two attributes if, for each value of the first attribute α there is more than one value of the second attribute β [Sao15].

Consider the following table representing students. It lists their department and their research areas.

id	**department**	**research_area**
20000001	ECE	Security
20000002	CS	Software
20000003	ECE	Circuits
20000004	SYDE	HCI
20000005	ECE	Networks
20000006	CS	Security

In this table there are two multivalued dependencies: *id* $\rightarrow$ *department* and *id* $\rightarrow$ *research_area*. To put this table into 4NF, we need to split it up into two tables, one with attributes *(id, department)* and one with attributes *(id, research_area)*. Multivalued dependencies are usually written $\alpha \rightarrow\rightarrow \beta$.

The formal 4NF decomposition algorithm looks something like this [SKS11]:

1. Compute D^+ (the implied set of constraints for the schema)
2. Check all relations to see if they are in 4NF.
3. If all relations are in 4NF with respect to the constraints, the algorithm terminates.
4. Otherwise let $\alpha \rightarrow\rightarrow \beta$ be a multivalued dependency in the relation R_i that has two or more multivalued dependencies.
5. Split the relation into two new ones, $(R_i - \beta)$ and (α, β).
6. Go back to step 2.

We have likely thus far observed that the process of putting a relation into a higher normal form means we break things up into more, smaller tables. The trend continues into higher normal forms.

The fifth normal form, as described in the MariaDB documentation, is one where you can't make tables any smaller with different keys. People have also tried to design sixth normal forms and there's some talk in the literature about domain key normal form but these are mostly of theoretical or academic research interest only. So we will focus on BCNF and 3NF mostly, perhaps with a little bit of 4NF. Our next step will be to examine functional-dependency theory and reason a bit more formally about the subject.

11 — Decomposition: Functional-Dependency Theory

Functional-Dependency Theory

Earlier we talked about the closure of a set of functional dependencies. Recall from earlier that the closure contains all the functional dependencies that are explicitly satisfied as well as those that are logically implied. That is, if $A \rightarrow B$ and $B \rightarrow C$ then it is implied that $A \rightarrow C$ and this is true no matter how many steps are in between, because this is transitive. More formally, if the functional dependencies are $A \rightarrow B$ and $B \rightarrow C$ then we know that for two tuples t_1 and t_2 if $t_1[A] = t_2[A]$ then $t_1[B] = t_2[B]$ and $t_1[C] = t_2[C]$. The notation to show the closure of a set F of functional dependencies is F^+ as previously discussed.

It is unlikely, however, that we want to compute F^+ from the definition of functional dependencies; if F is large there are many rules and many implied rules and we have to construct every logically implied element of F^+ from first principles. Instead, we would like to use *axioms*, handy rules of inference, that allow us to reason about the dependencies in a simpler way. The first three axions are simple enough and are called *Armstrong's Axioms*, named after the person who came up with them [SKS11]:

- **Reflexivity**: If α is a set of attributes and β is contained within α, then $\alpha \rightarrow \beta$ holds.
- **Augmentation**: If $\alpha \rightarrow \beta$ holds and γ is a set of attributes, then $\gamma\alpha \rightarrow \gamma\beta$ holds.
- **Transitivity**: If $\alpha \rightarrow \beta$ holds and $\beta \rightarrow \gamma$ holds, then $\alpha \rightarrow \gamma$ holds.

These axioms are considered both sound and complete, because they do not produce any errors and they allow generation of F^+ given F. There are proofs of these properties, but we prefer to omit those and just focus on using them. This is a minimal set, though, and there are some rules that we can derive some more from Armstrong's rules that would be convenient shortcuts [SKS11]:

- **Union**: If $\alpha \rightarrow \beta$ holds and $\alpha \rightarrow \gamma$ holds, then $\alpha \rightarrow \beta\gamma$ holds.
- **Decomposition**: If $\alpha \rightarrow \beta\gamma$ holds, then $\alpha \rightarrow \beta$ holds and $\alpha \rightarrow \gamma$ holds (reverse of previous rule).
- **Pseudotransitivity**: If $\alpha \rightarrow \beta$ holds and $\gamma\beta \rightarrow \delta$ holds, then $\alpha\gamma \rightarrow \delta$ holds.

We'll work on the example in the textbook([SKS11]) which says that the relation r has the attributes (A, B, C, G, H, I) and the functional dependencies are: (1) $A \rightarrow B$, (2) $A \rightarrow C$, (3) $CG \rightarrow H$, (4) $CG \rightarrow I$, (5) $B \rightarrow H$.

Based on the rules that we have, what logically implied functional dependencies can we observe?

There are five:

- $A \rightarrow H$ which is found by transitivity ($A \rightarrow B$ and $B \rightarrow H$).
- $A \rightarrow BC$ which is found by the union rule ($A \rightarrow B$ and $A \rightarrow C$).

- $A \rightarrow BCH$ which is found by the union rule ($A \rightarrow BC$ and $A \rightarrow H$).
- $CG \rightarrow HI$ which is found by the union rule ($CG \rightarrow H$ and $CG \rightarrow I$).
- $AG \rightarrow I$ which is found by pseudotransitivity ($A \rightarrow C$ and $CG \rightarrow I$).

The full algorithm for building up the closure just uses Armstrong's rules and is as below. The algorithm terminates when there is nothing left to add, which is certain to happen because the worst case scenario is that everything is functionally dependent on everything else which would mean if there are n attributes, there are $2^n \times 2^n$ possible functional dependencies. Now, the algorithm [SKS11]:

1. The initial condition is that F^+ begins as F.
2. For each functional dependency f in F+ F^+:
   (a) apply the transitivity rule to f and add it to F^+
   (b) apply the augmentation rule to f and add it to F^+
3. For each pair of functional dependencies f_1 and f_2, if they can be combined using transitivity, add the newly created combination to F^+.
4. If anything was added in steps 2 or 3, go back to step 2; otherwise the algorithm terminates.

Closure of Attribute Sets. Suppose we have some attribute(s) α and we wish to determine if it is a superkey. The strategy we will use requires us to compute the set of attributes that are *functionally determined* by α. An attribute B is functionally determined by α if $\alpha \rightarrow B$. Then the simple algorithm requires us to compute F^+ as above and then take all functional dependencies where α is the left hand side and the union of the right hand side of all dependencies, but this is difficult and expensive because F^+ may be large. Instead, a more efficient algorithm follows [SKS11].

1. The initial condition is that α^+ begins as α.
2. For each functional dependency $\beta \rightarrow \gamma$ in F, if β is contained in α^+, then γ is added to α^+
3. If anything was added in step 2, repeat step 2; if nothing was added, the algorithm terminates.

Much like computing the closure of F, here we compute the closure of α.

Going back to the earlier example with the five functional dependencies, we would like to evaluate something to see if it is a superkey. Given the rules, is A a superkey? Let's go through the steps.

The initial condition is that A is in α^+ (the result). Based on the rules, we can look at the rule $A \rightarrow B$ and see that we can add to the result B, which is now AB. The next rule that says $A \rightarrow C$ allows us to add C to the result so ABC. The rules that require CG on the left hand side cannot be added, at least not yet, because CG does not appear in the result (C does, but not G). We can look at the rule for $B \rightarrow H$ and add H to the result meaning the result is $ABCH$. This is not the full set ($ABCGHI$) and therefore we conclude that A is not a superkey.

This procedure can be repeated for any individual attribute and we will quickly find that no single attribute is a superkey for this relation. This is perhaps somewhat obvious from our list of functional dependencies. If we look carefully at them we can notice that neither A nor G ever appears on the right hand side of any of the functional dependencies, but all the other attributes do. That gives us a hint that a superkey might be AG, and that assumption can of course be tested by following the rules as above.

If our candidate is AG as suggested, we start off by saying the result is AG. Then iterating through the functional dependencies: the first rule of $A \rightarrow B$ means we can add B to get ABG. The next rule of $A \rightarrow C$ lets us make the result $ABCG$. The dependency $CG \rightarrow H$ allows us to add H and get $ABCGH$, and the dependency $CG \rightarrow I$ means the result is $ABCGHI$ which encompasses all attributes and we can terminate the algorithm here; our guess based on the hints was a good one.

There are three ways we can use this attribute closure algorithm [SKS11]:

- As above, to test if α is a superkey.
- Check if a functional dependency holds (by determining if it is in α^+).
- As another way to compute F^+; we just compute the α^+ for each α and then combine them.

Canonical Cover

Suppose we have a set of functional dependencies F on a relation. Whenever a user wants to update the data inside this relation, the database system must check that all functional dependencies in F are still satisfied (i.e., no key constraints or other rules are broken) [SKS11]. It is worth noting that in real life the rules are only enforced if you actually tell the database about them. If an update or insertion is inconsistent with the established rules, it is not permitted to happen.

Rather than checking every single functional dependency exactly as it is written, which may have some redundancy, we would like to simplify the set to a minimal number of rules that has the same closure. For example, if the rules say $A \rightarrow B$ and $B \rightarrow C$ and $A \rightarrow C$ we can immediately identify that there is redundancy and we could remove the rule $A \rightarrow C$ and still get the same closure, because that rule is implied and need not be checked separately.

You could argue that this sort of thing should not be necessary: designers should not introduce redundant rules and if they do it is their own fault. That is nice, but we don't live in a perfect world. Database designers who sit down and work out functional dependencies in advance and draw diagrams and plan ahead will probably have few redundant rules. But that's not what happens a lot of the time, either the system is designed well and then grows to have duplicate rules over time, or, very commonly, people just create some tables without thinking about it much and "we'll fix it later if we have to" is the approach. But it's hard to fix later, and in the meantime, the database has done a lot of unnecessary work...

Formally speaking, an attribute of a functional dependency is *extraneous* (unnecessary) under the following two scenarios [SKS11]:

- A is extraneous in α if A is in α and F logically implies $(F - (\alpha \rightarrow \beta)) \cup ((\alpha - A) \rightarrow \beta))$.
- A is extraneous in β if A is in β and the set of functional dependencies $(F - (\alpha \rightarrow \beta)) \cup (\alpha \rightarrow (\beta - A))$ logically implies F.

The first scenario is about removing a redundant attribute A from the left hand side of an implication; the second is about removing it from the right hand side. An example of the first is if our rules say $AB \rightarrow C$ and $A \rightarrow C$, then we could remove B from the left hand side. For the other scenario, if $AB \rightarrow CD$ and $A \rightarrow C$ are the functional dependencies, then C is extraneous in the right hand side [SKS11].

To check if A is extraneous on the right hand side, the routine is fairly simple; remove A from the right hand side (β) from the rule that it is in (e.g., if the rule is $D \rightarrow AB$, replace it with $D \rightarrow B$) and then compute the canonical cover of α (in this example, D). If the canonical cover included A, then A was extraneous in β.

To check if A is extraneous on the left hand side, the idea is pretty much the same: remove A from the left hand side (α) and check if this reduced set still functionally determines β. So if the rule is $DF \rightarrow GH$, remove F and we are left with the uncertain rule $D \rightarrow GH$, which we need to test to be sure. We compute the closure of D without this modified rule and see if it includes GH (actually, all attributes in β). If it does, then F was extraneous in the left hand side.

The canonical cover for F is denoted F_c and it is a minimal representation of F. It requires two rules: (1) no functional dependency in F_c has an extraneous attribute, and (2) the left side of each functional dependency is unique. The second rule means we can't have two rules that say $A \rightarrow B$ and $A \rightarrow C$ respectively; we must combine them in one single rule $A \rightarrow BC$. By making the minimal set, it makes it easier to test if the functional dependencies are satisfied.

Let's do an example from [SKS11]: The schema is simple (A, B, C) and our functional dependencies are ($A \rightarrow BC, B \rightarrow C, A \rightarrow B, AB \rightarrow C$). Well, anyway, it is simple enough, we can use the union rule to combine the rule $A \rightarrow BC$ and $A \rightarrow B$ (pretty obvious). And then we can remove extraneous attributes to get this down to

$A \to B$ and $B \to C$. This is slightly redundant and we can figure out by transitivity that $A \to BC$. These rules are pretty easy to test and we prefer to test these two versus the original four. This is a pretty terrible schema, actually, because there are way too many dependencies and only three attributes.

Let's try instead a different one from [EN11]: our schema has more attributes and the functional dependencies are $(B \to A, D \to A, AB \to D)$. In this case, all the left hand sides are different, so we can't combine any that way. The right hand sides are all single attributes so we don't need to look for extraneous attributes there. But we should look at the left hand side. The only one that is not minimal there is the functional dependency $AB \to D$ and we would like to know if we can eliminate A or B from the left side. Looking at the rule that says $B \to A$ we can probably reason pretty well that if $B \to A$ then $B \to AB$ (which you can also reason as adding B to both sides of the rule $B \to A$ becomes $BB \to AB$ and BB is just B). So we can replace the one double-left-side rule with $B \to D$. Then we have a set that is equivalent: $(B \to A, D \to A, B \to D)$. There is some redundancy here, though, and we could find out that $B \to A$ can be eliminated, leaving us with $(B \to D, D \to A)$.

Because canonical cover is about finding a minimal equivalent set, it is not necessarily that there is only one correct answer. If there is redundancy, we may need to delete one of two attributes (but not both). Let's go back to the example from earlier with the three attributes. Our functional dependencies are still $(A \to BC, B \to C, A \to B, AB \to C)$. If we test $A \to BC$ we will find that both B and C are extraneous. This is normally not an issue, because we just test one of the two, and if it is redundant, we delete it and move on to testing other attributes in the modified set. But one person might choose to delete B and another person might choose to delete C. Neither choice is wrong, but they will produce two different canonical cover sets, both of which are equally correct. The more functional dependencies there are with redundancy to be eliminated, the more possible correct answers there are.

Lossless Decomposition

Decomposition is breaking up a relation into two or more smaller ones. The motivations for doing so are something we will come back to later on, but for the moment, just assume there are good reasons. A decomposition is *lossless* if no information is lost by splitting a relation r into smaller relations r_1 and r_2; if information is lost it is called *lossy* (and that is undesirable). The more precise definition of this is specified in relational algebra as $\Pi_{R_1}(r) \bowtie \Pi_{R_2}(r) = r$ (or in SQL, the natural join of the two tables with all attributes) [SKS11].

How can we tell if a join is lossless or lossy? You may have guessed just from the placement in the topics covered that this has something to do with the functional dependencies (meta gaming the lectures?) and that would be correct. If the decomposition is lossless then every functional dependency f in F holds in spite of the fact that the relation is split. But we can't really cheat and just "forget" to put in the functional dependencies to make it work, now can we?

Thinking carefully about the way the decomposition must work, there needs to be some attribute or attributes that link together the two tables. This does mean some data is duplicated. One relation needs to have some way of referencing another, something like an address that references a person meaning the address relation contains the identifier of the person.

The formal definition for a lossless decomposition says that one of the following must hold on the two relations: $R_1 \cap R_2 \to R_1$ or $R_1 \cap R_2 \to R_2$. That is to say, the intersection of the two relations most be sufficient to uniquely identify one of the two of them.

If we have employee with *id, name, street, city, province, postalcode* and we tried to split it into r_1 as *id, name*, and r_2 as *name, street, city, province, postcode*. Is this going to work? The intersection of these two is *name* and this is not a superkey for either r_1 or r_2 because names are not unique.

There is a formal way for testing that dependencies have been preserved. There is a computationally expensive algorithm outlined in the textbook, as well as one that is premised on the idea that if each functional dependency f can be tested on one relation only, then just test it on that relation and we're done. Unfortunately that last assumption is not necessarily true; it could happen that a functional dependency is across tables...

Our solution is then a modified algorithm that does not require us to compute the closure of F. The algorithm is as follows, executed for each functional dependency $\alpha \to \beta$ [SKS11]:

1. $result = \alpha$
2. for reach relation R_i in the decomposition
   (a) $result = result \cup ((result \cap R_i)^+ \cap R_i)$
   (b) if $result$ did not change at all, break out of the for loop

If $result$ contains all attributes in β, then the dependency is preserved. And this must hold for all dependencies $\alpha \rightarrow \beta$.

Which is explained in two key ideas as follows. The first idea is testing if each dependency $\alpha \rightarrow \beta$ is satisfied in F', where F' is the union of the *restrictions* of F. That is, when we take the part of F that can be applied to a relation R_i we get F_i and F' is the union of all F_is. We compute the closure of α under F' to see if that includes β. The second idea is to use a modified version of the attribute closure algorithm to compute closure under F' without actually computing F' which would be expensive. For full details, see [SKS11], but keep in mind this is polynomial time rather than exponential time like computing F^+ would be.

Algorithms for Decomposition

We have looked at very simple schemas (and probably will continue to look at very simple ones in this course). For those it is not too difficult to see how to get the schema into BCNF or 3NF or design it that way in the first place. But if we need to take a schema and decompose it, for a sufficiently complex schema, there are some algorithms that make this process consistent.

BCNF. To decompose to BCNF, the algorithm is as follows [SKS11]. The initial state is the relations that we have, and we also need to then compute F^+. Then the steps are:

```
result := {R};
done := false;
compute F+;
while (not done) do
    if (there is a schema Ri in result that is not in BCNF)
      then begin
              let α → β be a nontrivial functional dependency that holds
              on Ri such that α → Ri is not in F+, and α ∩ β = ∅;
              result := (result − Ri) ∪ (Ri − β) ∪ ( α, β);
           end
      else done := true;
```

BCNF Decomposition algorithm in pseudocode [SKS11]

The algorithm produces relations in BCNF and a lossless decomposition. This is because when we replace a schema R_i with $(R_i - \beta)$ and (α, β) then the intersection of these two relations is α.

3NF. If instead we wanted to put things in 3NF, it is a somewhat more complicated algorithm. It is also dependency preserving and lossless. It explicitly builds a schema for each dependency in a canonical cover, ensuring that each schema contains a candidate key. Unlike the BCNF decomposition algorithm, this builds the relations up rather than breaks them down. A formal proof of the algorithm below is in [SKS11]:

```
let F_c be a canonical cover for F;
i := 0;
for each functional dependency α → β in F_c
    i := i + 1;
    R_i := α β;
if none of the schemas R_j, j = 1, 2, ..., i contains a candidate key for R
  then
    i := i + 1;
    R_i := any candidate key for R;
/* Optionally, remove redundant relations */
repeat
    if any schema R_j is contained in another schema R_k
      then
        /* Delete R_j */
        R_j := R_i;
        i := i - 1;
until no more R_j s can be deleted
return (R_1, R_2, ..., R_i)
```

3NF Decomposition algorithm in pseudocode [SKS11]

12 — Storage and Block Structure

Physical Storage

Until this point in the course we have looked at databases more from the perspective of application developers: people who use the database for a specific purpose. This conceptual level has been useful for telling us about how to interact with databases and even how to (re)design data storage in the database to meet certain goals. That may have been entirely review for those who have significant experience with using databases. Now it is time to move into the implementation side of the database. Treating the database as some sort of black box may be convenient, but we are interested in knowing how it does what it does.

Our first foray into the behind-the-scenes will be about how the data is stored. This will call back to what you have learned in operating systems about memory and storage. If you need to, now is a good time to review that material to read up on it. Although we will give a quick summary.

The fastest form of memory is registers, storage locations within the CPU itself. Then there is cache (often several levels of it), main memory, and finally, stable storage (solid state devices, magnetic hard drives, optical storage, and tapes). Memory is a series of size/speed/cost tradeoffs, with faster memory being relatively small due to its cost. Your operating system course will have covered, probably at length, caching and replacement algorithms. To avoid repetition, we will not spend any time on that.

The most salient fact that we need to think about is that there is primary (volatile, that is, lost if power is cut) storage, also known as main memory, and then secondary (non-volatile) storage (whether SSD, hard drive, tape, etc, doesn't much matter right now). To actually do any of the operations we want to do in a reasonable time, the data must be moved from secondary storage to primary storage, and eventually if the data is to be saved/updated it has to be written back to secondary storage. Secondary storage is generally very slow compared to primary storage, and therefore how we store our data becomes important because it determines how many times we need to access the slow storage (and how much time we spend waiting for it).

It's not as simple, unfortunately, as just packing as much into a block of storage as possible. We will still want to do that (usually) but there is more to it than just getting six records into a space where five used to fit. What is perhaps more important is: which five (or six) records to put in this particular block?

Do we have to do this? Well, yes. Consider three reasons [EN11]: (1) databases are too big for main memory (usually); (2) power outages and crashes DO happen, so nonvolatile storage is necessary; (3) secondary storage is affordable.

Data is stored in files, and files correspond to blocks on disk. The *primary file organization* determines how the file records are physically placed on disk (in the file, in the blocks) [EN11]. This obviously determines how we can access them. This is part of what determining the primary key is about.

Remember from earlier that tuples of a relation as returned by a select query do not have a guaranteed ordering (unless there is an order-by clause). That is because they could be stored in a primary organization in multiple ways. Our options could be [EN11]:

1. **Heap File**: An unordered file where new records are just appended to the end.
2. **Sorted File**: The file is ordered based on a particular field (the sort key).

3. **Hashed File**: The file is ordered based on the result of a hash function applied to a particular field.

4. **B-Tree File**: The file is organized using a B-tree structure.

This suggests strongly that if the data is not in the right order to begin with (and probably it is not) then extra work is needed to sort it. That is correct. Efficient sort algorithms exist, to be sure, but an order-by clause has a cost. We will soon examine in more detail the idea of query evaluation and execution.

A *secondary organization* allows efficient access to the data based on other criteria. An index is an example of a secondary organization which allows us to efficiently access the data based on that criterion. If we establish an index on an attribute, such as province in an address, it allows us to efficiently find tuples matching a particular province (e.g., "SK"). In the absence of that index, to find all addresses matching that province, we might have to scan all tuples.

We'll come back to the subject or organizing records in files in the next lecture.

Placing Records on Disk

The database will store records (tuples, relations, metadata, etc) on disk in a file or multiple files, as one would expect. The file itself is partitioned into blocks that match up with the file system's underlying architecture. Blocks are typically 4 KB but size may vary. Records are of variable size because they are user defined: if a user defines a relation with six attributes, a tuple in that relation may be a record.

Let us imagine that a relation consists entirely of fixed length fields, so we can say that a tuple has a known size. If the sum of the fields' storage needs is, for example, 94 bytes, then we can fit $4096/94 = 43.57$ records inside a block. For performance reasons as well as to keep chaos down we will usually not allow a record to be split across two blocks. Thus we would round down and say that 43 records can fit in this block (at most).

A careful reading of the previous paragraph said that the relation consists entirely of fixed length fields. That is, sadly, not a safe assumption in real life. Although we can restrict most fields to some number (e.g., 255 character string, 30 digits of precision number, etc), the database can be used to store arbitrary data in a "long text" sort of format. It is not hard to imagine a scenario: if you wished to maintain a database of all e-mails sent and received, the body of the e-mail (as well as the attachments) are of arbitrary size. Those are often called "BLOB"s – Binary Large OBjects – because, well, that's exactly what they are.

The fixed length records are a good starting point and we will deal with that first. Variable length-records that fit within a single block are the subject for another series of allocation algorithms. Finally, if there is so much data (e.g., email attachments and content) that we cannot possibly fit it into a block, we will likely store the content elsewhere and just keep a pointer to it in the record [SKS11]. Remember that this is just about storage – from the perspective of the user, a select query that asks for the body of the e-mail would return a tuple that has the body of the e-mail as an attribute just as if it were any other (fixed or variable length) string.

Fixed-Length Records. We will assume for the moment that an individual file contains only one kind of record. If a record is of fixed length, then we can easily compute how long it will be in storage by simply summing up the length of the fields. The example in `dsc` shows a type "instructor" with attributes ID (varchar of 5), name (varchar 20), dept_name (varchar 20), and salary (numeric 8,2).[9].

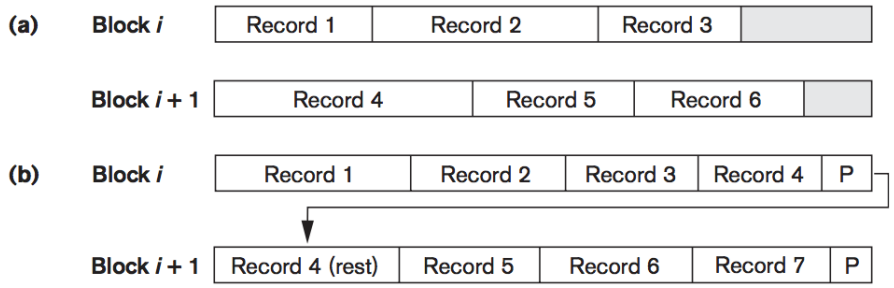
That's easy enough to sum up if each character is assumed to be 1 byte (so I guess this is not UTF-8) and we will assume the numeric 8,2 can be represented in 8 bytes (which is much more than needed since you'd actually really only need 5 to represent a number between 0 and 99 999 999.99, but let's assume that the type gets rounded up). That means the storage is $5 + 20 + 20 + 8 = 53$ bytes for this record.

That works. So the first 53 bytes are the first record, then the next 53 bytes are the second, and so on and so on and so on. 77.28 records would fit in a 4K block and we round down meaning 77 records fit in an individual block. The rest of the space is wasted, in some sense, in the same way that we may have internal fragmentation in memory allocation requests.

[9]This is super overly optimistic. Believe me, I am not getting paid eight figures a year here. Unless you count the leading zeros...

Even if we wanted to we would have trouble dividing such a record across two blocks: 0.28 (rounded to 2 decimal places; it's actually longer of course) of a record here is 15 bytes so do we put ID and then the first ten bytes of the name? Probably not. More likely is to have all of one field (ID) in block i and the remaining fields in block $i + 1$ which would reduce the amount of wasted space from 15 bytes to 10.

If we choose to allow a record to be in two blocks it is called *spanned* (see (b) in the diagram below), otherwise it is unspanned (shown as (a) in the diagram) (although this shows variable length records rather than fixed) [EN11]:



In any case it is undesirable to have the record split even if it can be split cleanly because it doubles the amount of work to read or write this record (because every operation is on two blocks). We have no choice if the record is bigger than a block, unfortunately, then we must use spanned organization.

A second problem is that it is difficult to delete a record in this structure. Either we have to have a way to mark the record as deleted or we overwrite it with some record from somewhere else in the file [SKS11].

If our initial state is as follows [SKS11]:

record 0	10101	Srinivasan	Comp. Sci.	65000
record 1	12121	Wu	Finance	90000
record 2	15151	Mozart	Music	40000
record 3	22222	Einstein	Physics	95000
record 4	32343	El Said	History	60000
record 5	33456	Gold	Physics	87000
record 6	45565	Katz	Comp. Sci.	75000
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology	72000
record 10	83821	Brandt	Comp. Sci.	92000
record 11	98345	Kim	Elec. Eng.	80000

And we choose to delete record 3, we can either (1) move each record up by one in the file to fill in that gap (left side in the diagram below); or (2) move the last record to fill in that space (right side in the diagram below). Both diagrams from [SKS11]:

record 0	10101	Srinivasan	Comp. Sci.	65000
record 1	12121	Wu	Finance	90000
record 2	15151	Mozart	Music	40000
record 4	32343	El Said	History	60000
record 5	33456	Gold	Physics	87000
record 6	45565	Katz	Comp. Sci.	75000
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology	72000
record 10	83821	Brandt	Comp. Sci.	92000
record 11	98345	Kim	Elec. Eng.	80000

record 0	10101	Srinivasan	Comp. Sci.	65000
record 1	12121	Wu	Finance	90000
record 2	15151	Mozart	Music	40000
record 11	98345	Kim	Elec. Eng.	80000
record 4	32343	El Said	History	60000
record 5	33456	Gold	Physics	87000
record 6	45565	Katz	Comp. Sci.	75000
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology	72000
record 10	83821	Brandt	Comp. Sci.	92000

The first approach requires significantly more work than the second: we have to continually copy each record and overwrite the previous one, potentially $n - 1$ times for a block containing n records when the first one is deleted.

If we move the last record into the open space it is less work but we still had to copy some data around. We could instead just mark that space as empty.

Marking the space as empty, however, has its own tradeoffs. If we want to insert a new record then we would have to scan through the whole block to see if there are any empty spaces. Instead of that, we might have a header structure in the file that tells us where the free sections are, if any, in a simple linked list or bitmap structure [SKS11].

Of course, things get complicated when we want to insert or delete records that are not fixed length. If we deleted record 3 in the previous example, what do we do if record 11 is bigger than record 3 was? What if it was smaller? We clearly need a different strategy for variable length records.

Variable-Length Records. As long as one field is of variable length then we have variable-length records in our file. There are some other situations that might cause record length to vary [EN11]: repeating fields, optional fields, or even putting multiple types of record in the same file.

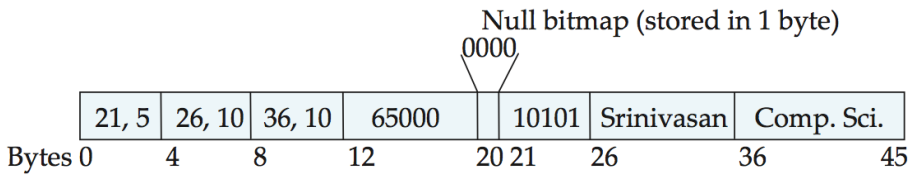
A repeating field is one that may have multiple attributes, such as an array of values. This seems strange at first glance since normally one might wish to have a separate relation to represent the array... On the other hand, if it is just a series of strings that might not be sensible. So a customs declaration with a list of clearance document numbers could have the array inside the declaration relation, or in a separate one.

Optional fields are not recommended. If fields are variable length, i.e., not fixed length, then an empty field does not take up any space and it isn't any problem. So let us perhaps leave that subject alone.

However the variable-length records are implemented, there are two operations we need to consider: (1) how to get records out of a block and (2) how to get particular attributes from a record [SKS11]. Because databases are for retrieval as well as storage we do need an efficient way to get data out again when we want it.

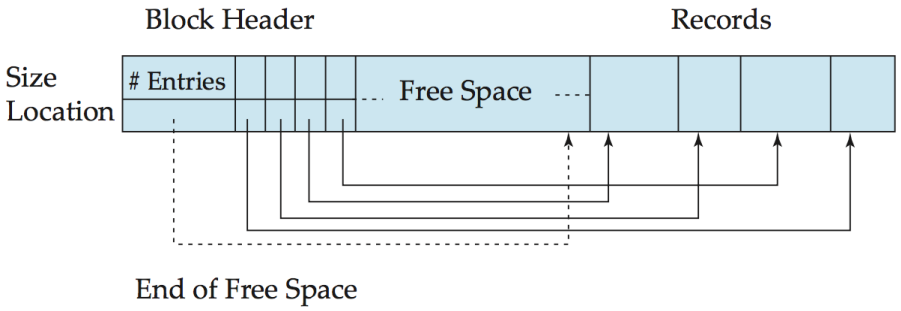
Variable length attributes are often represented in one of two ways. The first is a terminator character of some sort, something not normally allowed in the domain. The next is the attribute is prefixed with its length. If these options sound familiar, they should, because they are the same ways programming languages might represent a variable length string. As you will know, C chose the terminator approach (null-terminated strings) and other languages like Pascal chose the length prefix approach.

The sample layout structure from [SKS11] suggests that fixed length attributes are put together, then each variable length record has an entry of *(offset, length)* to explain where the variable length attribute is in the file. If the attributes ID, name, and dept_name in the previous example are variable length, and we have a fourth attribute of fixed length, salary (so it does have leading zeros!) then the length of the record can be computed. For the variable length record, if offset and attribute are 2 bytes each, then we have 4 bytes "overhead" per variable length attribute and we get something that looks like this [SKS11]:



The null bitmap (stored in one byte here), tells us which if any of the attributes have a null value. That is one approach to show what values are null; if a value is null when queried then there is no need to actually look at the stored data. The null bitmap could be stored at the beginning or it may not even appear at all: if an attribute is null its length can be zero and that is enough to know there's no data there. It may be a performance improvement to look at the null bitmap and decide if the rest of the data needs to be examined.

Then how do we put the variable length records into a block? For that, the strategy is the *slotted page structure*: a data structure header in each block. The header includes the number of records in that block, where free space ends, and then an array with the location and size of each record. This diagram shows what this looks like [SKS11]:



Records themselves are contiguous in the block and free space exists, perhaps counter-intuitively, between the end of the array and the first record. When a new record is added, it goes at the end of free space and the array gets a little bit bigger (so the free space is reduced at both ends) [SKS11].

Deletion requires a bit more work: if an entry is deleted the data is removed and any records between that record and the free space area are moved so that all free space remains contiguous. If a record needs to grow or shrink due to an update statement, then we have to repeat the same procedures and move the records around to ensure that all free space remains contiguous [SKS11].

Reorganization following deletion does not have to take place immediately; it is possible to do periodic reorganization when it is a convenient time.

Performance Considerations. We can see that there are performance implications for whatever implementation strategy we choose. Performance is not our primary consideration at the moment, nor is it a major focus of this course. However, it is a subject we will want to look at and think about on a regular basis.

A select statement can take longer for certain requests than others; if a record is in one block it doesn't take as long to retrieve it as it would if it is spanning two blocks. A seemingly simple update statement may take a different amount of time based on how many records need to be moved if variable allocation is used. If record sizes are fixed then assigning a value may take a constant amount of time... but the number of blocks we need to access to perform an update may be higher because fewer records fit in a block due to all the space being held in reserve for empty values that have not been filled.

13 — File Organization

File Organization

Recall from earlier that there were a few different file organization options we could choose [EN11]:

1. **Heap File**: An unordered file where new records are just appended to the end.
2. **Sorted File**: The file is ordered based on a particular field (the sort key).
3. **Hashed File**: The file is ordered based on the result of a hash function applied to a particular field.
4. **B-Tree File**: The file is organized using a B-tree structure.

We did not go into the details of the options earlier, but we will now be able to do so. Under most circumstances one relation corresponds to one file. However, multiple relations can be in the same file if desired. To start with, we will just have one relation in a file.

Remember that these file organizations are not affected by whether records are of fixed or variable length. Similarly, whether spanned records is allowed or not does not affect the file organization. Those sorts of consideration are about how to pack records into a block; this is about the order of the records. The first k records may be stored in the first block, but which records are the first k?

Heap File

The simplest way to order records is: don't order the records! The heap file has no inherent organization, although some (weak) idea of ordering may exist in the sense that records appear in the file in the order in which they are inserted. However, there is no guarantee about that, and if that is the behaviour it is an implementation detail and should not be relied upon.

This strategy makes insertion efficient – just tack this on to the end of the file, a constant time ($O(1)$) operation . If a new block is needed, allocate that new block and write it to disk. If not, the last block of the file is read, modified, and written. That part is simple. However, searching for a record is more difficult, because the file is not ordered in any way. Thus, we would have to perform a linear search ($O(n)$) over all records. To find one individual record in the relation it be necessary to examine, on average, half of the entries (assuming that all requests are equally likely). If the element is not in the file at all then we have to examine all the elements, and it was all for nothing anyway [EN11].

To delete a record, we need to first search for the record (linear search), then rewrite the block that it is in. Ultimately after enough deletions there will be multiple partially-full blocks, leading to wasted space. Thus, compaction of some sort will eventually need to take place which moves the records around to fill in these holes.

An update has all the problems we've discussed before. We must linear search to find the record(s) to the updated. It may lead to reorganization of a block or having to move the file to a different block, treating it as if it is an insertion and deletion.

Heap organization throws into the light the idea that the file structure has performance implications: that is, no matter what we choose, it will mean some operations are faster at the expense of other operations. Heap organization means that insertion is fast but reading, updating, and deleting are slower. This seems incongruous with our usual expectations of what databases are for. Normally we would expect retrieval to be the most common operation, right?

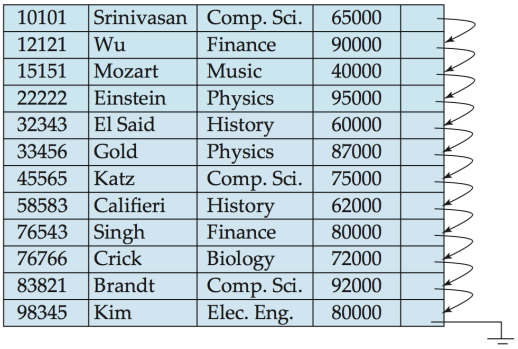
Sorted File

Long, long ago in an introductory programming course you learned about sorting, and perhaps more importantly, the idea that if you are going to search your data (more than once anyway). Usually in such sorting exercises you are asked to sort a bunch of integers. Integers have nice properties including a logical, simple ordering, and you can tell quickly at a glance if the array has been sorted correctly or not. In a sequential file, we will sort the records based on one particular attribute from start to end.

The field that we sort the file on is called the *ordering field*, and if it is a unique key then it is called an *ordering key* [EN11]. It does not have to be the primary key. A search that is on the ordering field is quite efficient. Suppose employees have sequential unique employee ID numbers and that is the ordering key in the database. If the goal is to find all employees with an ID less than 385, the data is sequential in all the blocks and that is convenient.

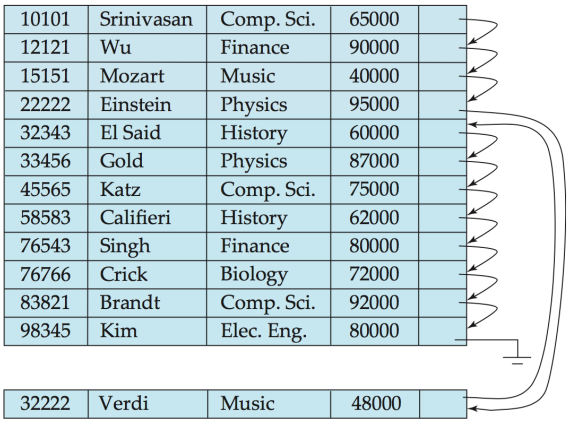
To find an individual record we can use a binary search ($O(\log n)$) which is for finding a record as well as update and deletion as discussed earlier. We are likely to access $\log_2(n)$ blocks, whether or not we find what we are looking for.

A simple sequential organization looks like [SKS11]:



In looking at this diagram we can see that they are sorted by some sort of integer ID (the leftmost column) in ascending order. And yet, there is a pointer in the rightmost column pointing to the next record and the next and the next. What purpose would that serve? It is for efficiency reasons, of course.

It can be difficult to maintain the physical ordering of the data because every insertion, deletion, and perhaps update requires a lot of moving records around. So we could potentially relax the constraint of having to keep all the records in order at all times. Instead, we could temporarily have the following physical layout [SKS11]:



If the insertion we are doing goes at the end of the file such as in the example an ID of 99001 is inserted, then the file maintains its sequential order. If there is a free space in the block where the record belongs, then we can put the record in the correct block although it is out of order. If, as in the case shown above, the block is full then we need to place the new record somewhere else. For that there is often an *overflow block* where records that do not fit into their normal locations go.

Deletion is the mirror image of insertion. If we deleted the record at the very end of the file, there is nothing special to maintaining order. If the record being deleted is in an overflow block, that makes the order more sequential rather than less. If the record inside a block is being deleted, such as the record with ID 58583 then the pointer from record 45565 would just be updated to point to 76543 and that would suffice. The empty space would be noted for the future.

If this continues over time, the file loses its sequential ordering as more and more records end up in overflow blocks. The file becomes less and less sequentially organized, meaning it resembles the heap file more and more. Eventually the file must be reorganized which sorts the records based on the sequential order again, potentially moving a large number of records to get them into the order where they belong. Because this is a potentially expensive operation, it would likely only be done when the system is not busy and when there are enough records out of order that it makes sense to do this.

Hashed File

A hashed file works very much like the sorted file, except, instead of the sorting key being a number, a hashed value is used instead. The hash technique can allow for fast access to records. The hash field is often the key. On its own that does not sound any different than the sorted file, but the "magic" in the use of the hash function is that instead of the ID being just a number, the hashed value tells us the address of the disk block in which the record is stored [EN11]. So instead of having to binary search to find the record with ID 99901, we could instead compute the hash value and find out that it is in block x and then go directly there.

We will take a short digression to cover the subject of hashing, actually. Chances are you covered this in a data structures and algorithms course, but it was probably in the context of hashing a value to find what index in the array to insert a particular element. That is *internal hashing*, which we will quickly review, but hashing can be much more.

If you implement a hash table simply, there is an array with M "slots" and then the hash function turns a particular value into an integer between 0 and $M - 1$. The hash function can be as simple as some integer field modulo M. That algorithm will work but probably does not provide "nice" properties to the hash function, or its output. If you don't have an integer field you want to hash (no "id" or other such field), don't despair. You can simply interpret everything as an integer if you want! It's all ones and zeros when you drill down to it and you can just pretend this string or double or binary data is an integer number and problem solved [EN11].

It is more likely that you actually want a hash function that has some better properties. A *folding* hash function does some arithmetic operations – add up some numbers, reverse some positions, move blocks around, the like, and usually a modulo operation is the last step to get everything cut down to the right size.

Regardless of how the hash function is calculated, we need to deal with the fact that two input values will eventually produce the same output value. This is called a *hash collision* and it is super common, because the range of output addresses is very often smaller than the range of input.[10] You may, for example, have an array that can contain 4096 elements, but if the input values are a range from 0 to 2^{16} then there will definitely be collisions. That itself is not a bad thing, as long as we have a way to handle this situation. We can apply three major approaches for how we handle hash collisions from [EN11].

The first is *Open Addressing* – simply go to the next open space if we must. Suppose the hash function computes n but there is already an element stored at that position. Then just go to position $n + 1$ and put it there (if possible, otherwise go to $n + 2$, etc). This is simple, but it can lead to clustering, where a bunch of elements are close together.

The next strategy is *chaining*. In this case, we have overflow locations. If there is a collision, we put the new item in an empty space and put a pointer from one element to the next.

The third strategy is a second hash function (*multiple hashing*) that is run if we have a collision to differentiate. We could have a third, fourth, etc. function but as one point we will default to open addressing because we will have no other choice.

A goal of hashing is to distribute the records relatively evenly across the address space so that we do not have too many collisions, but also do not have too much empty space. Ideal is something like 70-90% full; and if we need to make the table bigger (or smaller), then we do so. If M is a prime number, we get good distributions, although other technical constraints like the hashing algorithm may mean a power of 2 is best [EN11].

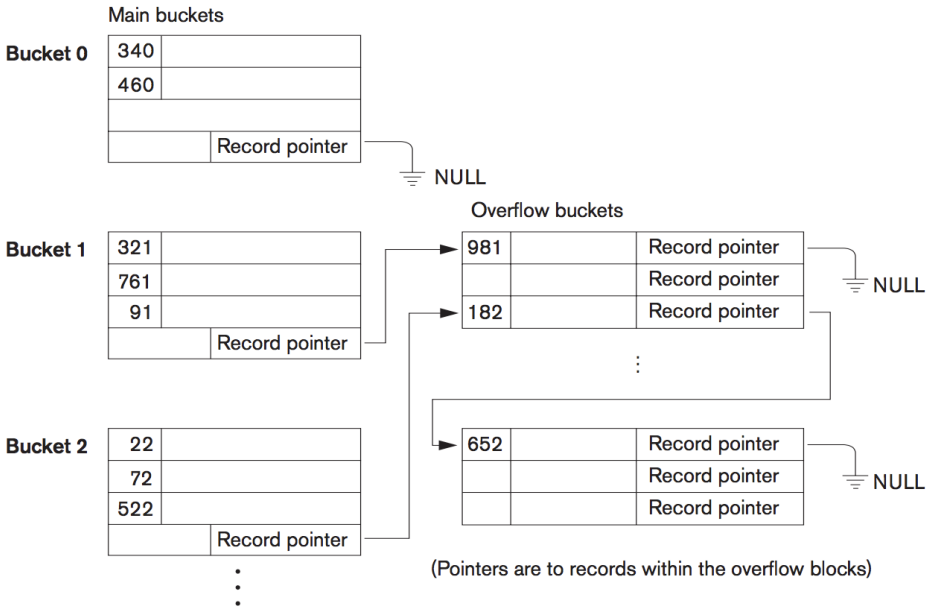
With the review of internal hashing aside, we should take a look at external hashing, which is what we need for files on disk. Instead of viewing the data storage as an array where we can store one element, instead we have buckets that hold multiple records... which is exactly what you should expect given that we have already talked about blocks. Then, the hashing function produces a bucket (block) number, and then from there we can get to the actual record by looking in the header for the block.

This does ameliorate some of the collision problem; if we have multiple records that produce the same hash value it's not a big deal because they can go in the same bucket. Nevertheless, we need to still be concerned about what happens when buckets are full, and as you might expect the solution for this will be overflow blocks.

Still, all the hashing we have covered so far is *static hashing*, that is to say, we have a fixed number of buckets. That is not suitable for our purposes a database where files will need to grow as the amount of data in the tables increases (or not be allocated as ridiculously large and mostly empty in the first place). If most buckets are close to full, we could simply add more buckets and then use a new hashing function (with the new maximum size) to reassign all the items to their new buckets, but this sort of global reorganization is very very slow [EN11].

The following image shows how a strategy of buckets with overflow buckets may look (simplified) [EN11]:

[10] There are "perfect" hash functions that don't produce collisions, incidentally.



Much better would be a hashing function that permits dynamic file expansion. That is to say, the file grows and shrinks as needed based on the amount of data that is in the file.

The first approach is *extendible hashing* – an access structure is stored in addition to the file. The access structure is an array of 2^d bucket addresses, where d is the global depth of the directory. The first d bits of a hash value are the index into the array to determine which element of the array is selected; the array element is the bucket in which the corresponding entry is stored. What makes this dynamic is that there may be duplicates in the array. Local depth d' ("Dee prime") specifies the number of bits on which the local bucket contents are based [EN11]. The hash value will usually be interpreted as a sequence of bits.

This may make more sense if displayed visually [EN11]:

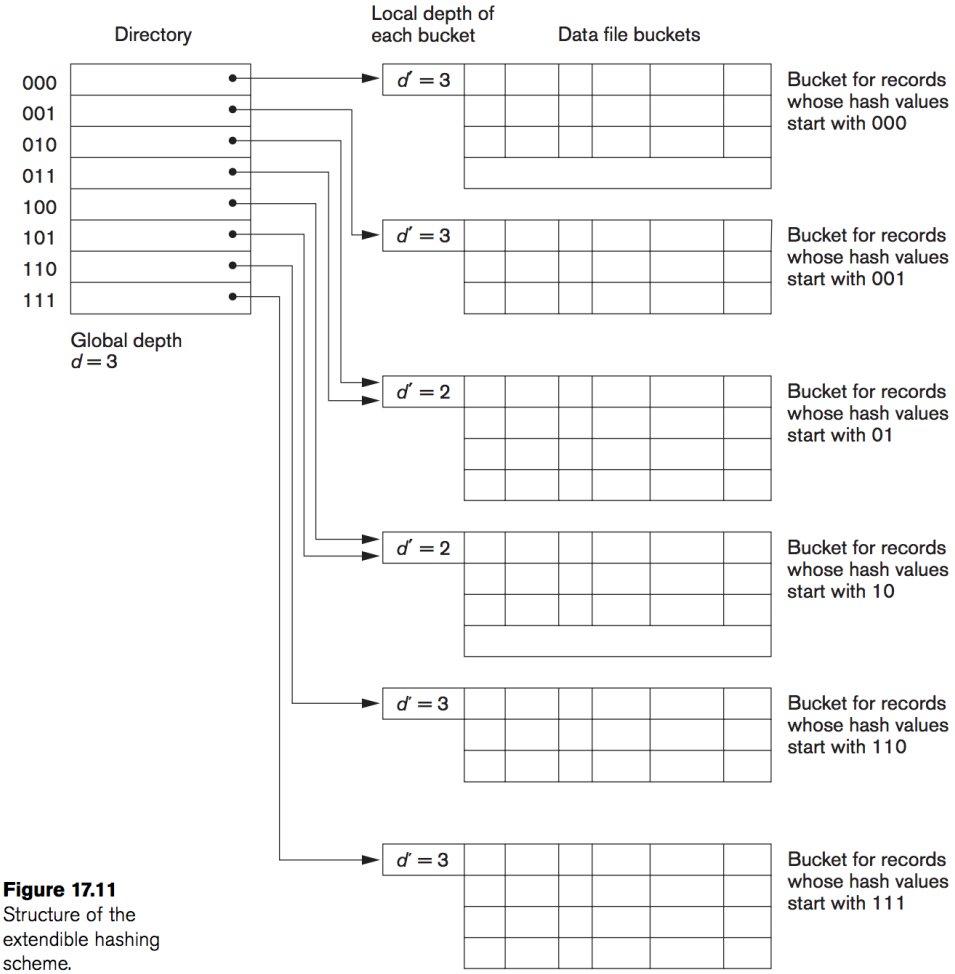


Figure 17.11
Structure of the extendible hashing scheme.

Suppose we are running out of space: that means that a bucket whose depth d' is equal to d has become full and is now overflowing. That would mean we increase the value of d which doubles the number of buckets. A quick example of bucket splitting from [EN11] in the previous diagram would be what happens if the bucket whose hash values start with 01 becomes full. At that point we need to split the records up between those that start with 010 and those that start with 011. Those are now two distinct buckets with d' of 3. That is fine, because d' can range from 1 to d (inclusive) but no higher.

If one of the buckets with d' equal to d becomes full, then we need to split those as well and the size of d must increase (in this example, to 4). Alternatively, if d' is smaller than d for all buckets (which could happen after enough deletions occur), we could potentially decrease d by 1 [EN11].

Extendible hashing is great because it allows the file to grow and shrink as is appropriate with the amount of data that is in the file. As things get close to full, we just create a new bucket (or several buckets) and the rate of collisions falls again. If we have to split a bucket, then a relatively small number of records need to be moved, because it is only some of the records in the bucket being split that need to go to the new bucket.

If we have to increase the value of d then we have a potentially larger and more painful reorganization, but we probably won't lose any sleep over that. Why not? When d is small, the total number of records in the system is not very large so the time to move all of them (worst case) is not very big. For example, if d is 3, if the table is completely full, there are $2^3 = 8$ buckets. Not very many. If d is large then increasing it is very rare. If d is 16 there are $2^{16} = 65\,536$ buckets and it takes a long time to fill them up. Ah, the power of exponential growth.

There exist other hashing strategies, dynamic and linear hashing, notably, but we will not examine them at the moment.

B-Trees

We will assume that everyone is familiar with B-Trees from a data structures and algorithms course (and perhaps a review from an operating systems course...). We will actually look into B-Trees in more detail when we talk about the idea of indexing. But theoretically we could use B-Trees to store the data in a way that very much resembles how operating systems store files on disk.

Multiple Record Types

Everything else so far has assumed that each type of record has its own file. That is not a requirement; it is possible to keep multiple types of record in the same blocks. This could speed up join queries that go over both tables but perhaps slow down other queries. To make this work we need a differentiator, of some sort, to tell us what kind of record each record is, and that is generally a simple field of some sort.

In a similar but less extreme variation, we could put tables that we know to be related next to one another on disk which could potentially reduce the amount of effort to read and write related records.

14 — Buffering and Indexing

Buffering and Caching

Buffering and caching are important in computing, as you surely know, since you have likely learned about it in multiple scenarios before now. The most recent probably related to caching in operating systems, where you may have modelled putting things into the L1 cache of a CPU. It is, nevertheless, applicable to databases as well. Where before we were concerned about whether we needed to fetch a page from memory into cache, now it is about whether we need to fetch a block from disk into memory.

As discussed, a block number is used for a read or write operation, which we can translate into an address for the disk operation. Ideally, the block is found in the buffer, because that would be faster. If the requested block is not there, we must load the block from disk, a very slow operation. Buffers are limited in size because memory is limited and it is expensive. At the time of writing, 32 GB of RAM is comparable in price to 8 TB of disk storage, so the money that buys 1 byte of RAM gets you about 250 bytes of hard drive storage.

As you can imagine, we must manage the buffer carefully, just as cache has to be managed to figure out what blocks should be in it. The what will make the biggest difference is the replacement algorithm: how we choose what blocks to be replaced. That's a subject we know something about since we have already covered it in the operating systems course.

There are two considerations that change the scenario a little bit from the operating system view: pinned blocks and forced output of blocks [SKS11]. *Pinned blocks* are necessary in the database: these are blocks that we do not allow to be written out to disk for some reason. The most common explanation for why disks cannot be written out to disk is that they are still being updated due to some transaction, and partial state should not be written to stable storage. *Forced Output* of blocks is the mirror image of that; it's writing a block out to disk even though we do not need the space it's currently taking up. This is often done to save the data to stable storage to minimize data loss in a crash.

If a block has been altered in the buffer, then that change has to be written to disk at some point. It can be done immediately when the block is changed, or it can be done when the block is evicted from the buffer. The second option means fewer main memory accesses, if a block is written to multiple times before it is sent to main memory. If a block has not been modified in buffer, it can simply be overwritten. If all other factors are equal, we should replace a block that has not been modified, as the work to write it out to memory need not be done.

Replacement Algorithms

The first approach to page replacement algorithms likely discussed was First-In-First-Out (FIFO). First-In-First-Out is quite easy to understand and implement. If there are N frames, keep a counter that points to the frame that is to be replaced (the counter ranges from 0 to $N - 1$). Whenever a page needs to be replaced, replace the page at the counter index and increment the counter, wrapping around to 0 where necessary.

The least recently used (LRU) algorithm means the page that is to be replaced is the one that has been accessed most distantly in the past. You might consider time stamps and searching a list, but because there are only two operations, it need not be that complex. When a page in the cache is accessed, move that page to the back of the list. When a page is not found in cache, the page at the front of the list is removed and the new page is put at the back of the list. This requires nothing more than a cyclic doubly-linked list.

Probably you have also learned that the LRU algorithm is the best choice. While this is reasonable in an operating system, which uses the past accesses to predict the future, the database may be capable of making predictions about what it going to happen, allowing better results. If the user requests an operation that will access all blocks of a record, e.g., a large select query, then before any blocks are loaded into memory we have a pretty good idea which ones we will want: those belonging to the relation being queried [SKS11]. In contrast, the operating system just sees instructions as they are issued by the program as a "surprise"; it has no predictive capability.

Suppose then that there is a request for a select query that covers all tuples of a particular relation. Something as simple as summing up the salaries of all the employees ("what is our payroll this month?") requires us to examine every tuple once and only once. That means once all the records in a block of the employee relation have been examined and added to the sum, we have no further use for that block and it can be replaced immediately. In the textbook ([SKS11]) this is called the *toss immediate* strategy.

Now imagine that we need to do a join query, such as selecting from addresses joining with employees. Now, we need to consider each block of employees for a given address (to see if there is a match and if the tuple should appear in the output). When we have finished looking at a block of employee records, it will not be looked at again until all other employee records have been examined [SKS11]. That is to say, after a block is used, it will not be needed again for a long time... which is the exact opposite of the assumptions for the LRU strategy to make sense. So what should we do instead?

The answer is: MOST recently used (MRU). When we are finished with a block, since we don't need it for a long time, we choose it for replacement. This strategy must be combined with the pinning strategy so we do not evict from the buffer the block we are currently still working through.

Is MRU the perfect choice? No – there is the optimal algorithm. The optimal replacement algorithm is fairly simple: replace the element that will be used most distantly in the future. For each block, make a determination about how many instructions in the future that block will be accessed. The selected block is the one with the highest value.

Unfortunately, there is a glaring flaw in this algorithm: it is impossible to implement. It requires clairvoyance (seeing into the future), and at least as far as I know, nobody has invented a way to do so reliably. The program and operating system have no real way of knowing which blocks will be used in the future.

As it is unimplementable, it is mostly a benchmark against which other algorithms can be compared. If we know that a given algorithm is, say 1% less efficient than the hypothetical optimal algorithm, then no matter how much we improve that algorithm, the best performance increase we can get is 1% [Tan08].

Since we cannot predict everything that will happen in the future, we can just look at usage patterns for the past, compare them to the optimal algorithm and choose whether LRU or MRU or something else is best. Then hope that future usage patterns look a lot like past ones.

In addition to pinned blocks, there is also the forced output of blocks. Sometimes the crash recovery subsystem insists that certain blocks be written before the requested write can take place [SKS11]. That is something that we will come to in the future.

Another small exception to our strategy for block replacement is index blocks. Since we will likely access the index frequently over the course of any operation, we want it to stay in memory. That does assume our query makes use of indices of course. But the index blocks idea leads us into a discussion of the index in general.

Indexing

The concept of an index is familiar to anyone who has read a textbook or other large volume. At the beginning of the book there's an index (the "table of contents") and this can be used to quickly get to where you want to go in the book. If you want to learn about Indexing and Hashing, then you find this entry in the index and it tells you that what you are looking for is on, for example, page 475. With that information in hand you can get to the chapter you want very quickly. If this index did not exist then the only way to find what you were looking for is the hard way: search through the book until you reach that chapter.

There's actually an even better way in most large books. The table of contents is one sort of index, but if you

wanted to find out about the subject of Indexing and Hashing you had to linearly search the table of contents. At the back of the book there is a more detailed listing of the topics found within, this time by keyword of the name. This list is sorted alphabetically, meaning that it is much quicker to find a particular keyword because you can binary search this index.

If the file already exists then it has some sort of organizational structure as we have previously discussed, even if that organization is no specific order. As we have already seen, if we want to find an record in such a file, we have no choice but to perform a linear search. The index saves us from doing that... if what we are looking for has an index.

Not all attributes in a relation will have an index, nor should they. Creating and maintaining an index takes significant work (and some space as well as other incidental costs). And it makes sense to have an index for things we expect to search. If you are searching for a book, you can search by title or author, for example. It might be sensible to have an index on those fields. If you wished to search for books published in a certain year, say, 2016, and there is no index on that field, we must scan the entire table to choose what tuples go in the result relation.

But let's assume that the operation is on a field that has an index so the index is useful. The index data will be smaller than the data file (unless we somehow have defined an index on every field, but... just... don't do that) and it can be kept sorted which means that it is quick to binary search the index and that tells us efficiently where we need to go to get to the desired record.

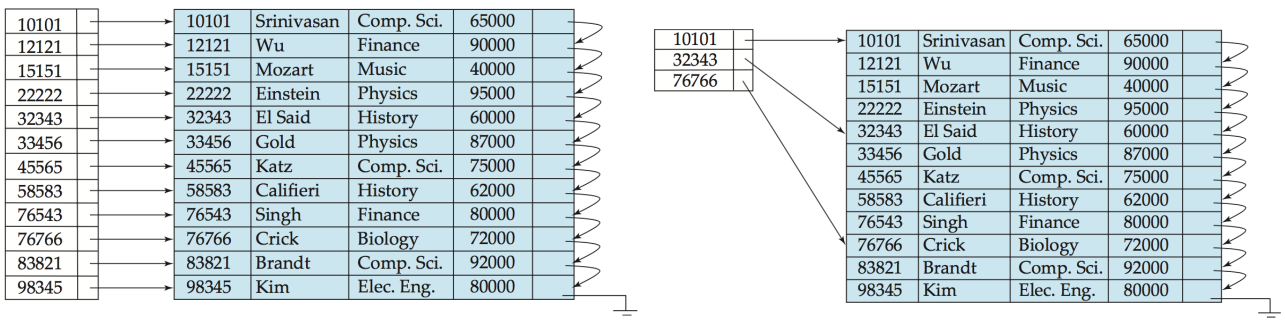
There are a few types of ordered index as defined in [EN11]. The first is a *primary index* and this is the ordering key field of an ordered file. This is the physical ordering of the file, but if the ordering field is not unique, then a *clustering index* is used instead (which makes this a clustered file). A *secondary index* is an index on any field other than the one(s) used for ordering the file.

Primary Indexes. The primary index is a file itself, which itself is a listing of tuples. Each tuple contains the key and the address of the block where the data is stored. So if there is an address with id `24601` and it corresponds to disk block x then the entry for this in the index is $< 24601, x >$.

One question we will face is whether every single tuple in the relation should have an entry in the index. If we choose yes, it is called a *dense index*; otherwise it is called a *sparse index*. A dense index is boring and works exactly how one might expect; there is an entry for every single entry. A sparse index has relatively fewer entries which means searching it is faster to search and operate on.

To use a sparse search index, we find the index entry with the largest search key value that is less than or equal to the value we are looking for then we go to the record pointed to by that index entry and then search in that block until we find a match (or can conclude that there is no match). This has the advantage that the index file is significantly smaller, but it will be slower to return an empty result if we do not find the value sought. It does, however, mean there are not quite as many entries that all point to the same block.

An example dense index (left) and sparse index (right) in on the same data is shown below [EN11]:



A small drawback to primary index has to do with re-ordering when insertion and deletion take place.

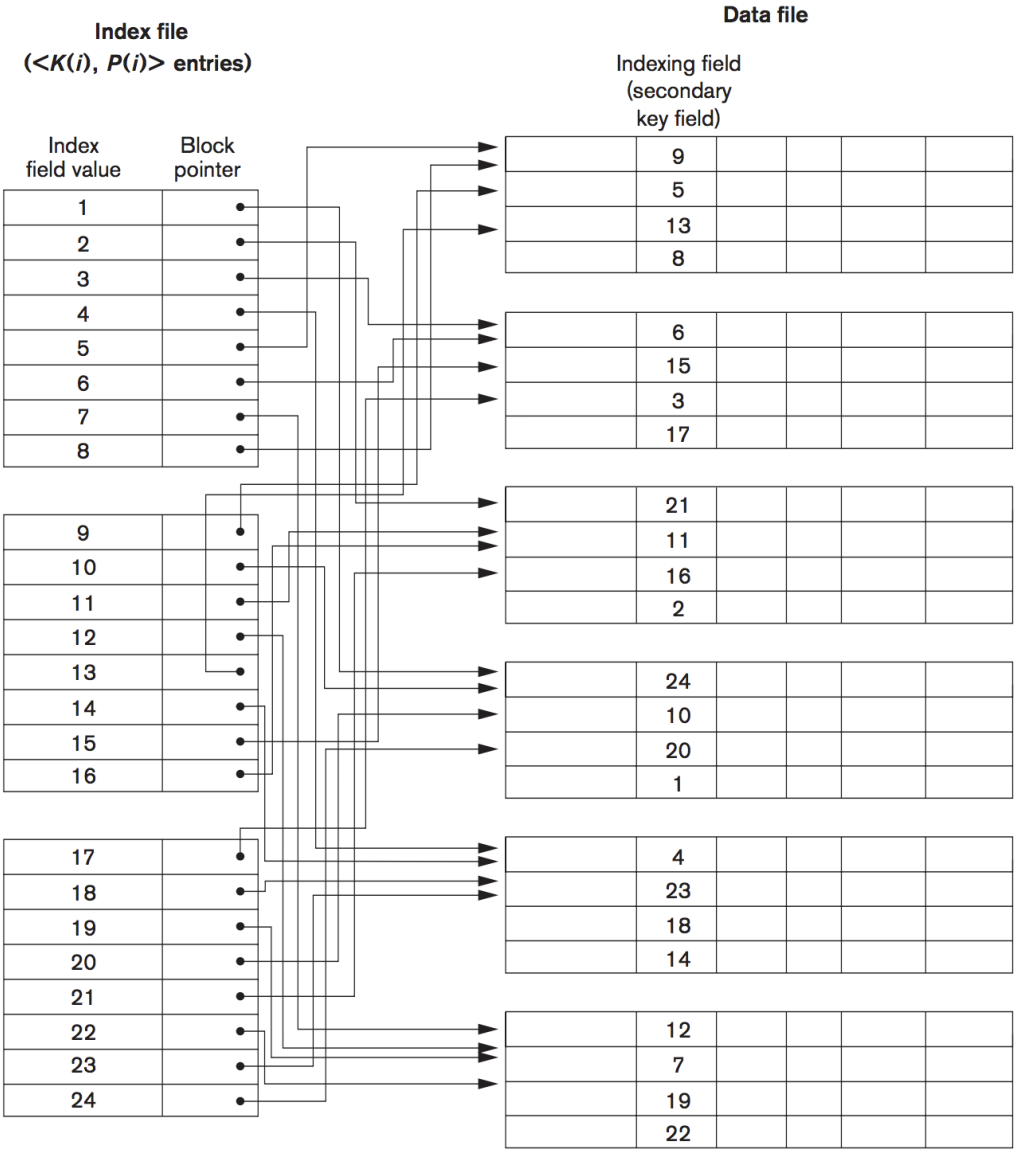
Clustering Indexes. Suppose instead that records are ordered on a non-unique field. That means there is a clustering index and the clustering index is a lot like a sparse index. There is an entry for each of the distinct values

of the ordering field. As a possible performance enhancement, to avoid the problem of insertion and deletion, one might have a block (or several) for each of the unique values of the ordering field [EN11].

As we generally discourage the use of non-unique fields in place of a primary index this scenario will not receive a great deal of attention.

Secondary Index. A secondary index, by definition, does not map to the order of the file. An index can be created on a unique field or a non-unique field and this, unlike clustering, is encouraged. As always, the index has two fields, the the indexing field and a block pointer (or record pointer).

If a secondary index is on a unique field, there is one index for every record in the data file and the secondary index will be dense. We cannot leave things out, because the file is not ordered based on the attributes being sought. An example of what a secondary index would look like is shown below [EN11]:

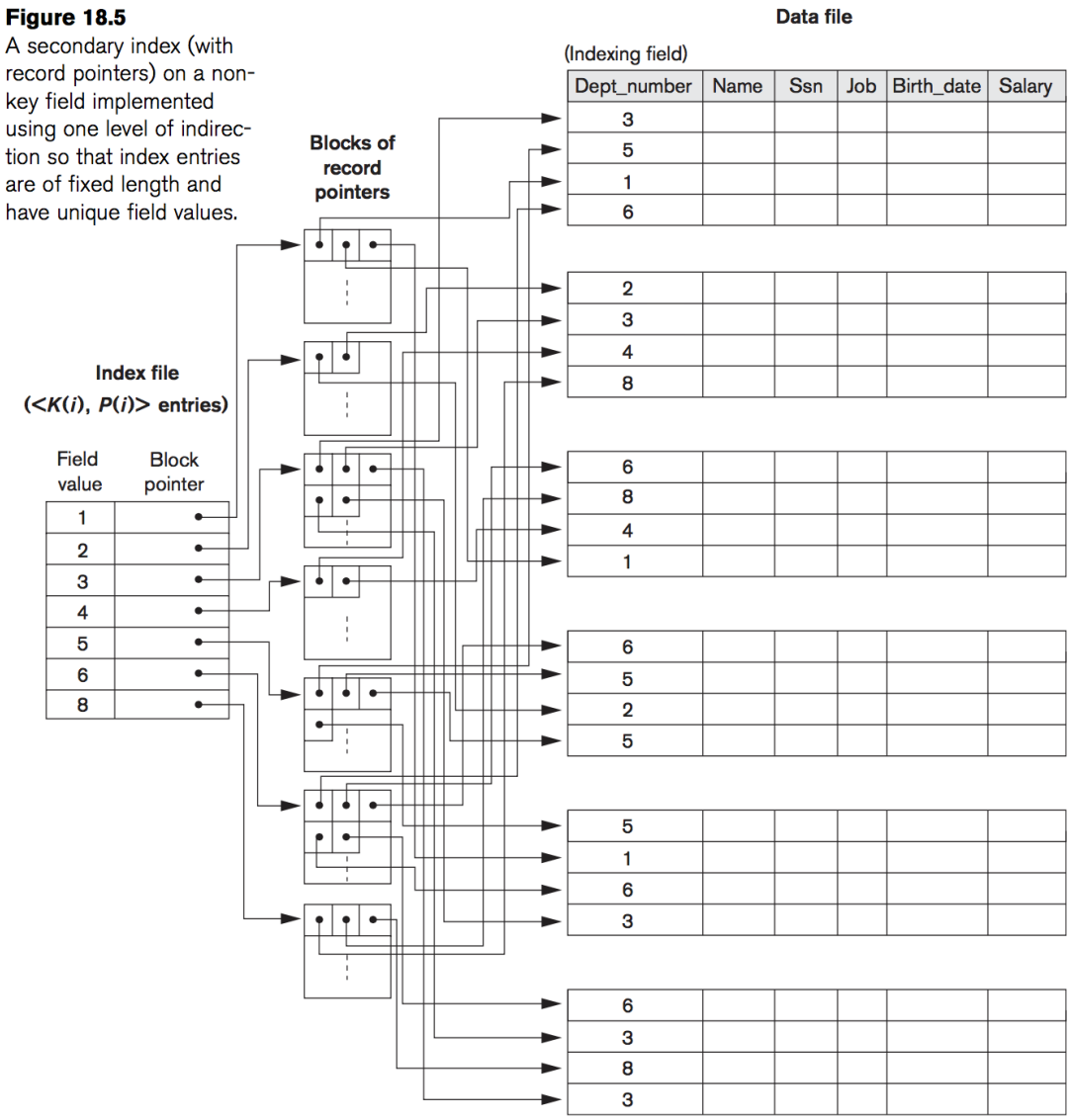


Of course, a secondary index does not have to be on a unique field. But that increases the complexity slightly. We could choose any one of these three options [EN11]:

1. **Duplicate entries** – allow multiple index entries with the same value, maintaining a dense index. Our search routine must account for the fact that there could be multiple results with the same value. This would permit $< x, a >, < x, b >, < x, c >$ in the index.

2. **Variable-Length Entries** – entries can have variable length, so the tuple $< x, a >$ can now expand to be $< x, (a, b, c) >$. Again, the search algorithm must accept the fact that there are multiple elements on the right hand side.
3. **Two-Level Index** – There is an additional level of indirection so that the tuple references another block; this second block references the data blocks themselves.

The two level index approach is very common. It is shown below in the diagram from [EN11]:



Regardless of what implementation is used, operating on a secondary index is slower than operating on a primary index. That's okay, because it is still much better than not having an index at all, which would force linear searching.

Next Steps. The secondary index has provided quick introduction to the idea of an index with multiple levels. The next topic will dig more into the subjects of multi-level indexing and the use of B^+ Trees

Updating an Index

Regardless of the type of index, whenever a record is inserted or deleted, the index needs to be updated. If an update affects an indexed field, then it must also be updated. Calling back to updating a relation, an update can be modelled as a deletion and an insertion, which means we no longer need to consider update separately.

We will therefore consider two simple algorithms for insertion and deletion from [SKS11]:

Insertion. If the index is dense, then:

1. If the search key does not appear in the index, insert an index entry at the appropriate position.
2. If the search key is found then:
   (a) If the index stores pointers to all records with the same search key value, add a pointer to the new record in the index entry.
   (b) Otherwise the index entry stores a pointer to the first record with that value; place the record being inserted after the other records with the same key.

If the index is sparse then:

1. If a new block is created, insert the first search key value from the new block into the index.
2. Otherwise, no new block is created and:
   (a) If the new record is the lowest record in the block, update the index entry for that block.
   (b) If the new record is not the lowest, no change to the index occurs.

Deletion. For deletion, the system must search to find the item to be deleted. If it's not there, of course, there's nothing to do.

If the index is dense, then:

1. If the deleted record was the only one with its value, delete that search key from the index.
2. Otherwise:
   (a) If the index stores pointers to all records with the same search key value, delete the pointer to the deleted record.
   (b) Otherwise: if the deleted record is the first in the group, update the pointer to be the next item; if it is not then there is no change to make.

If the index is sparse then:

1. If the index does not contain an entry with the deleted value, there is nothing to change.
2. Otherwise:
   (a) If the deleted record is the only one with its search key, replace it with the next search key and record pointer; unless that is already in the index, in which case just delete the entry.
   (b) If it is not the only one with its search key, then update the pointer to point to the next record with that same value.

These algorithms are generally extensible to multi-level indexes as well: update the lowest level index and the changes will cascade upwards if needed. From the point of view of the second level, the lower level is the same as if it is a file containing records. But, as we will see, this is something we are about to examine.

15 — Multilevel Index: B+ Trees

Multilevel Index: B^+ Trees

If instead of having just a flat index file, we decide to have multiple levels, then it means instead of performing a simple binary search on the index, we must traverse a tree. At first that may appear counterproductive, or at least, not helpful, since we would still have to traverse a bunch of blocks to get to the item we would like to find. That is true if we pick a binary tree, but, as the title of this section gives away, that's not what we will do.

Thinking back on how linear search works, we check an item to see if it matches and we reduce the remaining space to check by 1. Not efficient. If we do a binary search, we reduce the remaining space to check by half of its current size. Much better. Hypothetically we could improve on that if we could reduce the remaining space to search by more than that.

This is called the *fan out* of the multilevel index, and instead of dividing the remaining space to search by 2 like in binary search, we divide it n-ways where n is the fan out. Then we reduce the record space by a lot. Searching the multi-level index requires about $log_{fo}($ number of blocks $)$ which is an improvement if the value for fo is larger than 2, which it almost certainly is. In fact, we would like to see the fan out is in the order of 50 - 200 [EN11].

If our fan out is five, for example, when we need to find some data then it means we can at each step eliminate 80% of the remaining search space, which will obviously bring us to the answer much faster than if we eliminated only 50% of the search space. This pattern can be repeated at multiple levels of the index, which allows us to narrow in on the target quickly.

A common organization for files is called ISAM – Indexed Sequential Access Method. The IBM version of this has a two level index which maps as closely as possible to the hard drive underlying structure [EN11]. MySQL has its own implementation called MyISAM but it is terrible... Please do not use it; use InnoDB instead.

Let us examine a very simple approach for multi-level indexing. The lowest (or first, or base) level consists of blocks that contain pointers to the second level. The second level has blocks that contain pointers to the actual file.

Whether the file has multilevel index or not, in an index sequential file the performance does get slower as more entries are added. Reorganization can help, but it is expensive. Instead we might wish to use a balanced tree structure, specifically, a B^+ tree. As you have learned about the subject in a data structures & algorithms course in the past (and likely in an operating systems course too...), we will have just a quick refresher on how it all works:

A B^+ Tree structure, formally, has the following characteristics [Sta14]:

1. The tree is made up of nodes (have children) and leaves (have no children).
2. Each node contains at least one key identifying a file record, and more than one pointer to child nodes or child leaves.
3. Each node has some maximum number of keys.
4. The keys in a node are stored in non-decreasing order.

And a B^+ Tree of degree d has the following properties [Sta14]:

1. Every node has at most $2d - 1$ keys and $2d$ children ($2d$ pointers).
2. Every node other than the root has at least $d - 1$ keys and d pointers. Each internal node except the root is at least half full and has at least d children.
3. All leaves appear on the same level.
4. A non-leaf node with k pointers contains $k - 1$ keys.

To find something in a B^+ Tree, the general algorithm is fairly simple [Sta14]:

1. Start at the root node.
2. If the key is in the current node, it is found, and the algorithm terminates.
3. If the key is less than the smallest key in this node, follow the leftmost pointer; go to step 2.
4. If the key is greater than the largest key in this node, follow the rightmost pointer; go to step 2.
5. If the key is between the values of two adjacent keys, follow the pointer between them; go to step 2.

Insertion into the B^+ Tree is more complicated, because of the rules that keep the tree balanced [Sta14]:

1. Search the tree for the key. If it is not found, then we are at least looking at the block where it would be if it were there.
2. If this node has fewer than $2d - 1$ keys (that is, it is not full), insert the key into this node in the proper sequence. The algorithm terminates.
3. If the node is full, split this node around the median key into two new nodes with $d - 1$ keys each. Promote the median key to the higher level. If the new node is less than the median key, insert it in the left-hand new node; otherwise into the right.
4. The promoted node is inserted into the parent node in order, splitting the parent if the parent is already full.
5. If the process of promotion reaches the root node and the root is full, then splitting and promotion occurs and the height of the tree increases by 1.

Deletion is a little bit similar to insertion (but a little bit complicated) [SKS11].

1. Search the tree for the key. If it is not found, then there is nothing to do and the algorithm terminates.
2. Remove the target (and remove it from the parent nodes if it appears at intermediate levels as well), shifting the remaining entries if needed to ensure there are no gaps.
3. If the current node is the root and there is only one remaining child, then the child is the new root and delete the current node.
4. Otherwise, if the current node is less than half full then:
   (a) If an immediately adjacent node is exactly half full, then coalesce the current node with that adjacent node and update the pointers of the parent.
   (b) Otherwise, redistribution is necessary: move an item from the adjacent node so that the current node is at least half full and update the pointers of the parent.

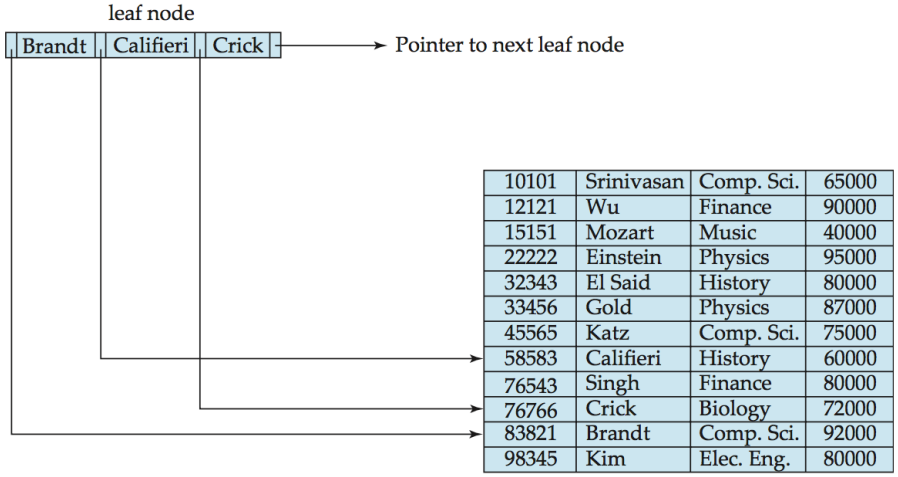
This glosses over, to some extent, how the actual deletion and insertions are performed. In particular, deletion requires some thought as to how to actually write the algorithm. The textbook contains a more extensive algorithm in pseudocode for both insertion and deletion.

There are some differences versus a general B-Tree which are worth calling out. We require that leaf nodes be at least half full (unless it is the root node and there isn't much in the file). The last pointer in a leaf node chains to the next leaf node to allow relatively quick access if we need to move through the data sequentially. So the last pointer is reserved and can't point directly to data. Non-leaf nodes are sometimes called internal nodes. Unlike your typical B-Tree, the items in the non-leaf nodes are duplicates (repeats) of the ones below. Why? The pointers in the non-leaf nodes point to leaf nodes, not the actual data.

Why do we like to use trees? Well, here's an example from [SKS11]. Assume a node is 4 KB, we are searching a field of length 32 bytes, and pointers to disk blocks are 8 bytes. This means we can fit about 100 entries in a block. If the file to be queried contains 1 million records, then a lookup requires only $\lceil log_{50}(1\ 000\ 000) \rceil = 4$ accesses. This is the worst case, though, since the root node may be in the buffer frequently due to its frequent access. If we had to do this with a regular binary tree, the $\lceil log_2(1\ 000\ 000) \rceil = 20$ accesses and that is also the worst case for binary search. It is true that insertion and deletion are computationally expensive, but since the disk is the slow part of the system it is preferable to spend more CPU cycles computing the rebalancing of the tree than it is to spend more time accessing the disk.

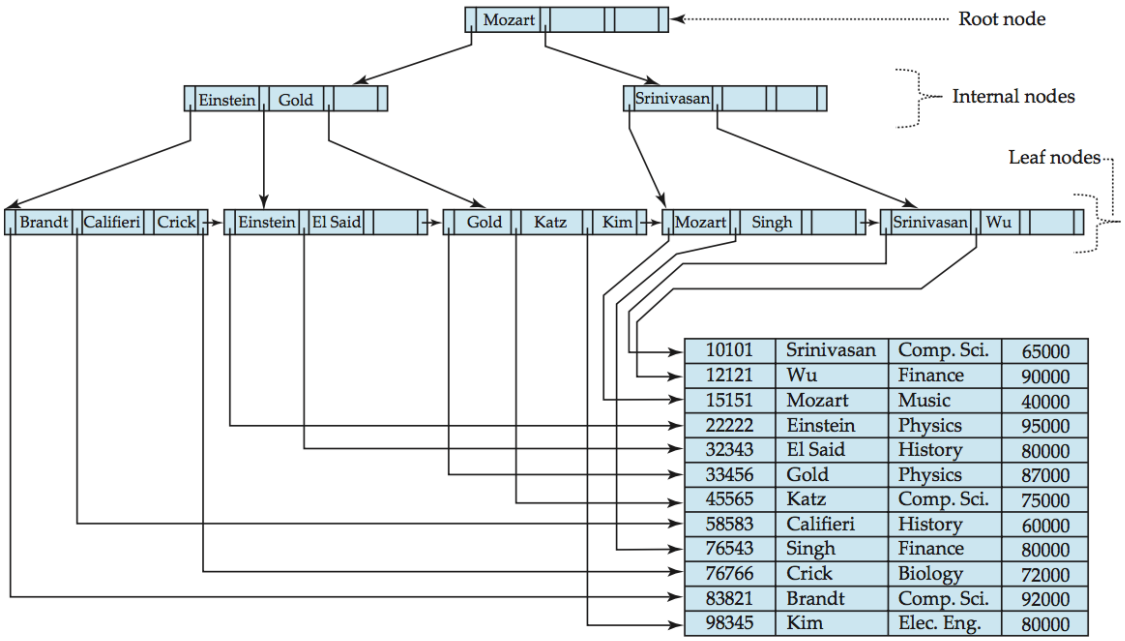
Overall it seems like the benefit of much faster access to data outweighs the negatives of the B^+ Tree structure: duplicate data, more computational complexity to insert/delete, et cetera.

And now let us see an example of applying that to the "instructor" relation. First, a leaf node [SKS11]:



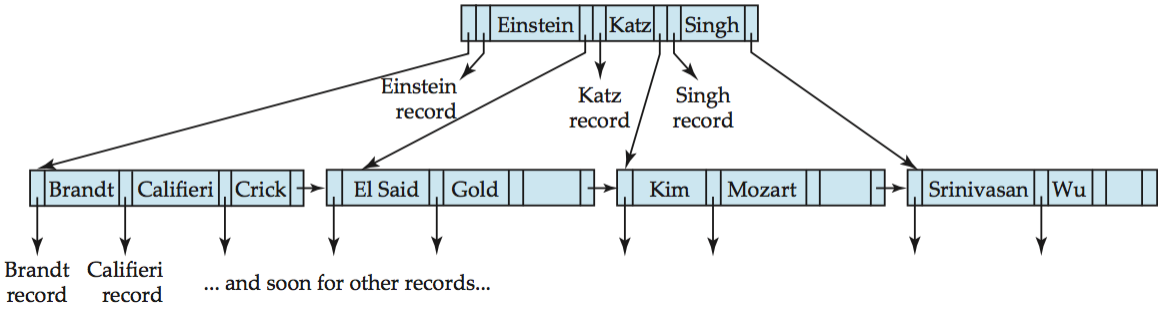
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	80000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	60000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000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

And then the complete tree for the instructor relation [SKS11]:



Even though name is non-unique (there's a lot of people with the name Wang/Wong, according to Wikipedia, in 2007 there were almost 93 million in China alone...!) this could easily be a secondary index and that would probably make sense.

Just for our interest, if this were a regular B-Tree and not a B^+ Tree then the storage would look something like this [SKS11]:



Small Considerations in Indexing

Indexing Strings Strings can cause a bit of a small problem when it comes to B^+ Trees because they can be of variable length or rather long, causing a pretty "tall" tree and losing the benefits of the structure. The suggested solution in [SKS11] is called prefix compression: store a prefix of the search key value that is sufficient to distinguish the sub-trees. Thus we could store in a non-leaf node "Jans" instead of "Jansen" if the closest two subtrees are under the subtrees are on the left "Janet" and on the right "Janson" (note different spelling) [SKS11].

B^+ Tree File Organization Just as we have made use of a B^+ Tree organization for the index file, the actual file containing records could be stored using B^+ Trees as well.

Covering Index. A *covering index* is one that stores the value of some attributes (other than things we search on) alongside the pointers to the record [SKS11]. This increases the size of the index, but allows us to fulfill some queries without having to even go to the data blocks of the file at all. The advantage, of course, is that this can be much faster – if the data we need is at hand in the index, we just take it and use it and we need not load the

record from disk. The downside is that it increases the size of the index, means fewer items can fit in one index block, and requires additional overhead to keep it up to date.

To give an example, it might be sensible to have in the covering index for employees, their last name alongside the regular index search key (ID). This would mean that queries where only the name is required, or name and ID, don't need to go into the file blocks at all.

There are limits to the size an index entry can be, so it's not possible (or sensible) to have all fields of a record in the index entry. Even if you could, it would pretty much defeat the point of having index blocks at all...

Multiple Key Indexes

All discussion so far has covered the simple scenario where one index on one attribute is being used for a query. However, that is not the only way. Sometimes an index will be constructed on two attributes, or, we can combine two (or more) indexes on a single attribute.

Indexes on Multiple Attributes. Let us imagine that we have a set of addresses. It is likely that we may want a composite key on the attributes of (city, province) since city names can be duplicated: there is a Waterloo in Ontario and there is a Waterloo in Quebec (to say nothing of Belgium...). If such an index is defined, then the search key in the tree structures will be the combination of both attributes.

This is helpful in situations where we know that we are going to search on particular combinations of a certain attribute frequently. The city-province example is just one of many where we are likely to perform queries that hit both attributes. Another example might be first and last name.

It may help to think of composite keys as just sort of smashing together the two attributes (perhaps with some separator like a dash) so you end up with Waterloo-ON as the key. From there it works mostly like the regular search.

Assuming there is no index on city alone, if we wanted to find all addresses with the name Waterloo, this is possible. If the select query says to select all addresses where city equals "Waterloo", our search term in the tree structure is then something like "Waterloo-%". Seems simple enough.

Combining Multiple Indexes. Let us imagine that employees have an index defined on the fields ID, department, and salary. If we wish to do a select that finds the IDs of all employees with the department name "development" and salary above $100 000, there are three strategies for carrying out this query [SKS11]:

1. Use the index on the department to find all records for the development department. Check each record to see if the salary is above $100 000.
2. Use the index on salary to find all records for employees with a salary above $100 000. Check each record to see if the department equals development.
3. Use the index on the department to find all records for the development department. Use the index on salary to find all records for employees with a salary above $100 000. Compute the intersection of the pointers to find out which ones satisfy both criteria.

Which approach is best? It depends a lot on our data, but at a first glance only the third option makes use of both indexes, but it does require the computation of the intersection. The first approach may be very efficient if the number of employees in the development department is quite small. Similarly, the second approach is quite efficient if the number of employees making over $100 000 is quite small. The third approach is perhaps best if neither the department nor the number of people with a large salary is small...

Even then, the third strategy may be a poor choice if there are a lot of employees in development, and a lot of employees with a salary about $100 000, but only a small number that fulfill both criteria (and do you really want to work there?). In that case we have to do a lot of work to produce a small number of results when one of the other approaches might have been faster.

This idea that there are three different ways to carry out the requested operation is actually an interesting and large topic. We will therefore take the next few lectures to discuss how the database server understands a query, optimizes it, and carries it out.

16 — Query Processing

Query Processing

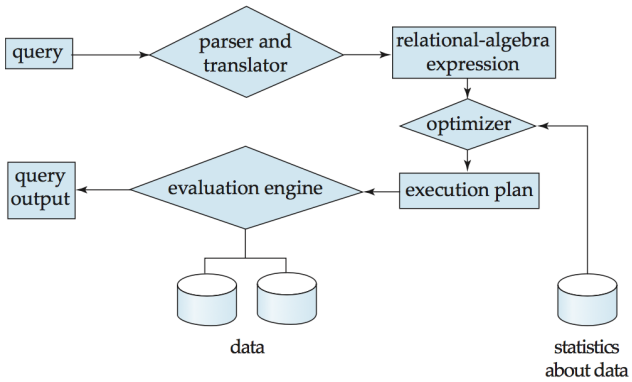
Imagine you are given an assignment in a course and you are going to do it now. To get the assignment done, you will probably (1) figure out what exactly the assignment is asking you to do; (2) figure out how you are going to go it (e.g., must do part 1 first because part 2 depends on it...); and finally (3) do it!

The procedure for the database server to carry out the query are the same [SKS11]:

1. Parsing and translation – interpreting the SQL query in a form the computer can work with.
2. Optimization – figuring out how best to carry out the query.
3. Evaluation – execution of the query according to the plan just developed.

To break these steps down into more detail, as in [EN11], the steps are a bit more like this. First, the scanner needs to figure out what are keywords and attribute names and relation names to figure out the text of the query. The parser will then check that it is valid SQL syntax. If so, then it will be validated to make sure that the attribute and relation names are valid inside the schema of the database being queried. Then this query is turned into a query tree or graph, which is used to devise the execution strategy. Then finally we can execute the query.

A quick visual summary of all the steps we are going to go through [SKS11]:



We will not spend time talking about the scanning, parsing, and verification steps of query processing. While they are all very important things and we need them, that is the sort of material that belongs in compiler textbooks and you may have already covered (or at least begun to cover) such material in a compilers course. In the meantime, if a query is syntactically invalid, or contains misspelled attributes, or something to that effect, we can't really do anything with it a query with an error is simply rejected and goes no further through the process. We therefore will give them no further consideration.

Usually a query is expressed in SQL and that must then be translated into an equivalent relational algebra expression. Complex SQL queries are typically turned into *query blocks*, which are translatable into relation algebra

expressions. A query block has a single select-from-where expression, as well as related group-by and having clauses; nested queries are a separate query block [EN11].

A query like `SELECT salary FROM employee WHERE salary > 100000;` consists of one query block. The relational algebra for this expression is pretty easy to work out, and it gives us two possibilities. We can select all tuples where salary is more than 100 000 and then perform a projection of the salary field of that result (e.g., throw away the fields we do not need). The alternative is to do the projection of salary first and then perform the selection on the cut-down intermediate relation.

Suppose there is a subquery, like `SELECT name, street, city, province, postalCode FROM address WHERE id IN (SELECT addressID FROM employee WHERE department = 'Development');`. Then there are two query blocks, one for the subquery and one for the outer query. If there are multiple query blocks, then they do not have to follow the same strategy; they can be optimized separately if desired.

The relational algebra is not quite enough to actually carry out the operation. What we need instead is a *query execution plan*[11]. To build that up, each step of the plan needs annotations that specify how to evaluate the operation, including information such as what algorithm or what index to use. An algebraic operation with the associated annotations about how to get it done is called an *evaluation primitive*. The sequence of these primitives forms the plan, that is, how exactly to execute the query [SKS11].

If there are multiple possible ways to carry out the plan, which there very often are, then the system will need to make some assessment about which plan is the best. It is not expected that users will write optimal queries; instead the database server should choose the best approach via *query optimization*. Optimization is perhaps the wrong name for this because we are not choosing the optimal approach; instead we will make some estimates about the query plans and try to choose the one that is most likely to be best. The subject of query optimization is something we will touch on now, but also come back to when we have more understanding.

Measures of Query Cost

If you are asked to drive a car from point A to point B and there are multiple routes, you can evaluate your choices. To do so you need to break it down into different sections, such as drive along University Avenue, then get on Highway 85, then merge onto 401... Each segment has a length and a speed, such as knowing that you will drive 4 km along University Avenue and it is signed at 50 km/h (although with red lights and traffic and whatnot the actual average speed may be more like 30 km/h). By combining all of the segments, you get an estimate of how long that particular route will take. If you do this for all routes, you can see which route is the best.

Of course, it may turn out that real life gets in the way: if there is a crash on the highway, traffic really sucks and your decision that taking this particular route would be fastest turns out to be wrong. Short of being able to see into the future, this is more or less inevitable: estimates are just informed opinions and things may be worse (or better) than expected.

Where does the time go in executing a query? The biggest component is most likely loading blocks from disk, considering how slow the disk operations are. Although we can improve this via use of SSDs, many large databases still use hard disk drives. CPU speeds are pretty fast, to the point even where memory accesses seem slow from the perspective of the CPU. So, no matter what we do, disk accesses likely account for far more of the time than CPU computation. In reality, CPU time is a nonzero part of query optimization, but we will ignore this (as does [SKS11]) for simplicity's sake and use only the disk accesses to assess cost.

The number of block transfers (data moved in and out of memory) and the number of disk seeks (repositioning where on the disk we are reading from) are the important measures of interest here. SSDs have no seek time and have faster transfer speeds, but the number of block transfers is still relevant. To compute the estimate of how long we think it will take to perform an operation, the formula is $b \times t_T + S \times t_s$ where b is the number of block transfers, t_T is the time to transfer, S is the number of seeks, and t_s the time it takes to seek [SKS11]. For a hard drive, transfer times are on the order of 0.1 ms and seek times are about 4 ms.

We will follow the estimating strategy in [SKS11] which comes with a few caveats, explained in this paragraph. Sometimes writes can be twice as expensive as reads. This is because the disk subsystem may read back the written data to check that the write succeeded, but we will assume that does not happen here. Similarly, at this first level

[11] `https://www.youtube.com/watch?v=fQk_832EAx4`, or `https://www.youtube.com/watch?v=l3FcbZXn4jM`

we are not including the amount of time it takes to write a final result back (that depends on the query... a select, for example, may not need it, where an update will). That cost would be accounted for separately. And the asterisk here is that the size of memory makes a big difference – if the buffer is large enough that all of the database fits in it, then disk reads are almost never needed. We will imagine the worst case scenario, that is, only one block per relation can be in memory at a time. If we are "wrong" and the data we need is already in memory, the actual cost is less than the estimated cost (which is better than the reverse).

The estimates calculate only the amount of work that we think it will take to complete the operation. Unfortunately, there are several factors that will potentially affect the actual wall-clock time it takes to carry out the plan:

- How busy the system is – if there are multiple concurrent operations then any particular operation may be queued or blocked or otherwise not able to proceed immediately, leading to a longer time to completion.
- What is in the buffer – if partial data is in the buffer that will speed up completion of the operation since some planned disk operations can be skipped.
- Data layout – if the data is packed well on disk then we need to do fewer seek operations (or shorter ones, perhaps); likewise, if the data is distributed over multiple physical disks we can sometimes do some reads in parallel, but it's hard to know exactly how much parallelization is possible.

You can probably think of various other factors that will affect the time it takes to execute the query in reality. The estimate is just an educated guess that we will use to plan how to do the work; it's not a promise of exactly how long it will take.

Note also that the lowest cost approach is not necessarily the fastest. Sometimes we can go faster by using more resources, but the approach the database often takes is the lowest cost (specifically, fewest disk reads). Recalling the earlier driving analogy, you can think of this as perhaps driving a longer route that involves more highway driving and therefore less time, even if it means more fuel consumption due to the increased distance and speed. When driving, we generally prefer to choose the lowest time estimate, but there are also people ("hypermilers") who are really obsessed with getting maximum fuel economy... and the database is one of those people!

Select Operation

The first example we will see about estimating cost is how to do a select operation. We will consider a lot of possibilities, some of which are obviously sub-optimal. We should consider all options (this is good advice in science in general, consider all possibilities, even the null hypothesis) even if we are pretty sure one of them is going to be terrible... we might be surprised. In other cases, a seemingly-terrible algorithm is our last resort, such as if we have to find something that has no index, we perform a painstaking linear search. The strategies considered in this section come from [SKS11].

We will assume that we have a select-from-where condition to be evaluated on a single relation. The relation itself is stored in a single file and this file contains only records of that single relation.

General Selection

The first approach is general and really needs no introduction.

Select Strategy 1A: Linear Search. The linear search is exactly what it sounds like: scan each file block from beginning to end to see if the selection condition is satisfied. We can assume the file is stored efficiently so we have only one seek to start reading the file, and then the transfer time for each block. Thus there is one seek plus one transfer for each block of the file and the cost estimate is $t_s + n \times t_T$ where n is the number of blocks in the file (worst case scenario).

As mentioned earlier, linear search is probably inefficient if we have any other alternative. However, if there is no index and the file is not ordered on the search predicate, we may not have any other choice. But as a last resort, we can be sure this will work.

Selection with Equality

We would much prefer, if we can, to use an index of some sort. If we have that, we can access a lot fewer records than we would otherwise need to. The following group of strategies are applicable when the where condition contains an equality test, e.g., "province equals Ontario".

Select Strategy 1B: Binary Search Binary search is also exactly what we are pretty well familiar with. If the file is ordered on something other than the primary key, and this is the where condition, we can search using a binary search algorithm. Binary search would then mean we can significantly reduce the number of blocks we need. But we will assume generally from here that we will look at B^+ trees.

Select Strategy 2: Primary Index, Equality on Key If the select's where clause specifies equality on a primary index, i.e., the way the file is organized, it is very efficient. We will have to traverse the tree to get to the leaf containing the pointer to the record. That means one seek and one transfer for each of the levels of the tree. Then finally we access the data directly, which is another seek and another transfer. Thus the cost estimate is $(h_i + 1) \times (t_s + t_T)$ where h_i is the height of the index B^+ tree.

Select Strategy 3: Primary Index, Equality on Non-Key If the primary index for the file is a non-key (non-unique) field then instead of getting one record we may get multiple records. Of course, this should be a rare occurrence because the primary index being a non-key is not recommended. But if such a query has to be carried out we need an estimate of the cost. The multiple records will be consecutive, because it is the primary index. Thus the cost will be $h_i \times (t_s + t_T) + t_s + b \times t_T$ where in this case b is the number of block containing the target search key. Because the blocks with the actual data are stored sequentially we don't need to check the index again for the next ones, nor do we need to seek to them. But we do have the one seek to the first block.

Estimating b can be somewhat difficult. If we know there are two matches they could be in the same block or in two different blocks. We may need to guess about how many match.

Select Strategy 4: Secondary Index, Equality If the secondary index is on a unique field, then the cost estimate is the same as strategy 2: $(h_i + 1) \times (t_s + t_T)$.

If the key is not unique then the cost is higher. Unlike strategy 3, we cannot rely on the fact that the records are consecutive. In fact, they are almost certainly not. We make the opposite assumption, that each record requires a seek and a transfer of a new block. This the estimate is $(h_i + k) \times (t_s + t_T)$ where k is the number of records matching the condition.

As with b in strategy 3, estimating the value of k is probably difficult. In the worst case it means reading from every block of the file because there is (at least) one matching record in each block.

Selection with Comparison

Until this point we just handled an equality condition, where we looked for an exact match. A comparison is more complex. We can still use the linear search approach, if we must, but if we have an index we may still prefer to use that. We will continue to assume that the file index is organized in a B^+ tree.

Select Strategy 5: Primary Index, Comparison Suppose our desired search value is x. If the attribute being compared is A, then our possibilities are (1) $A > x$ or $A \geq x$, (2) $A < x$ or $A \leq x$.

For the first case, we use the tree to go to the first tuple where $A > x$ or $A \geq x$ respectively. Then we do a file scan from that tuple to the end of the file. In that case, our cost estimates are like those of strategy 3.

For the second case, we start at the beginning of the file and go until the first tuple with $A = x$ or $A > x$ respectively. This is more or less a linear scan that may end early. When doing this comparison, the index contributed absolutely nothing (although we could look at it, one imagines, to refine the estimate of where the reading will end).

Select Strategy 6: Secondary Index, Comparison This case very much resembles the case of strategy 4, equality on a non-key field. Fetching the actual data is a pain, of course. If the number of records to be fetched is large,

because of all the jumping back and forth, it might actually be worse than simply performing a linear search. It depends on the value of k, how many records will match.

Complex Selection

The previous strategies, aside from linear search, assumed the predicate had one clause. Before we start, a quick review of the ideas of conjunction, disjunction, and negation (from basic algebra) from [SKS11]:

A conjunctive selection involves the logical and operation. In relational algebra, the selection operation is written $\sigma_{\theta_1} \wedge \sigma_{\theta_2} \wedge ... \wedge \sigma_{\theta_n}(r)$. This describes selection of all the tuples that match all of the predicate clauses.

A disjunctive selection involves the logical or operation. In relational algebra, the selection operation is written $\sigma_{\theta_1} \vee \sigma_{\theta_2} \vee ... \vee \sigma_{\theta_n}(r)$. This describes selection of all the tuples that match one or more of the predicate clauses. This is equivalent to the union of all the tuples matching each of the individual θ_i conditions.

A negation selection $\sigma_{\neg\theta}(r)$ is the set of all tuples of r for which the condition θ evaluates to false.

Select Strategy 7: Conjunction, One Index Since we need to satisfy all conditions, we will first find all records that satisfy one of the simple conditions, and then throw away any of the retrieved that do not match the remaining conditions. One of the previous strategies (2 through 6) will suffice, and the cost is pretty much determined by which of those strategies is used. We would take estimates for each strategy and choose the one with the lowest expected cost.

Ah, but which simple condition do we choose? If only one has an index, then we use the one index. That was easy. If there are multiple choices, ideally we choose the most restrictive – the one that will match the fewest tuples. That would then result in less loading of tuples that we are just going to throw away. This is another case where, as you might imagine, it is not immediately clear how to know which condition matches the fewest tuples. There are approaches for making good decisions, which will be a topic of future discussion.

Select Strategy 8: Conjunction, Composite Index Recall from the discussion of indexing that a composite index is one that is on multiple attributes. If the selection has multiple simple conditions, a subset of which exist in a composite index on the relation, that is very helpful. Then we can use the index directly. If the complex where predicate is exactly equal to the composite index, we will be using one of strategies 2, 3, or 4. If there are some conditions that form the composite index and others that do not, our strategy looks more like strategy 7. Again, take some estimates of each possible execution plan to decide what to do.

Select Strategy 9: Conjunction, Intersection Another way we could do the conjunction would be to look at the record pointers. We get a list of record pointers for each simple condition that has an index and compute the intersection of these. If there are any conditions on attributes without an index, then we must further look through the results to throw away those records that don't meet all conditions.

The cost here is the sum of the cost of looking through each index, plus the cost of loading the records after the intersection is computed. In theory we can reduce the amount of work necessary by simply sorting the pointers before we retrieve data, which would make the disk reads closer to sequential and minimize the amount of seeking that needs to be done.

Select Strategy 10: Disjunction, Union Much like the conjunction strategy, we get a list of record pointers for each of the conditions where there is an index. Instead of computing the intersection, we find the union (no duplicates) of these pointers. Again, sorting may make the disk reads closer and speed things along significantly.

If there is at least one condition that does not have an index, then it is makes sense to just linear scan since we are going to have to do that anyway.

Select Strategy 11: Negation The authors in [SKS11], as they are known to do, leave the explanation of this strategy "as an exercise" for the reader. Although one can, of course, do a linear scan to find those that, unlike usual, do NOT meet the given conditions, what can one do?

If we have an index on the field that might help: we can evaluate the index and decide which records match the not condition, and only load those. As the last resort, the linear search approach will be necessary.

17 — Query Processing, Continued

Query Processing, Continued

Having examined in some detail the idea of different strategies for carrying out a select operation. The next thing we would like to do is expand to more advanced queries, specifically, join queries. Join queries will be cripplingly inefficient if they operate on unsorted data. Remember that in a join, we need to match a tuple of the left hand relation with a tuple of the right hand relation. If for every tuple we had to linearly search the right hand relation, that would be painful. One way or another, we probably have to sort one of the relations to make this work.

Digression: Sorting

So let us take a few minutes to talk about sorting. Wait, I hear you say, you already know ALL about sorting, insertion sort and selection sort and bogo sort and radix sort. Yes, also merge sort and quick sort. When you learn about sorting in a data structures and algorithms context, you are sorting a manageable amount of data. Manageable in the previous sentence means the full set of data can be fit into memory. For those, we can use all the standard algorithms. In the world of databases we need an algorithm that does not depend on loading everything into memory.

Sorting relations that do not fit into memory is called external sorting, and we will learn the external sort-merge algorithm from [SKS11]. The basic plan works a lot like the merge sort algorithm works: divide the data into smaller units, sort the smaller units, then merge the smaller sorted units into a larger sorted unit. The sorting of each smaller unit will take place in memory, as per normal, and then we need to do an N-way merge where N is the number of smaller units to be merged. Let's expand on this.

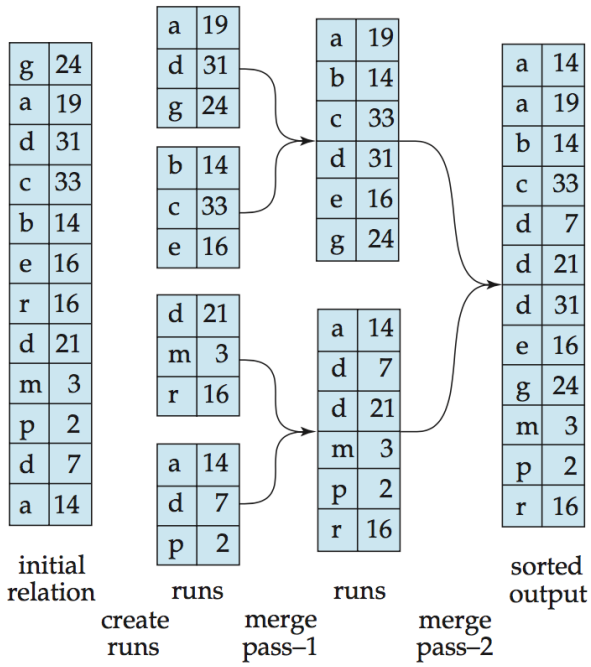
Step one of the algorithm is to divide the file into N chunks of size M where M is the number of blocks that can fit into the area of memory available for sorting. Each chunk is as big as it can be for the constraints of the system, but no bigger. This chunk is called a *run*. Each run i is then sorted and it is written to a temporary file called R_i.

Step two of the algorithm now merges it. If N is less than $M - 1$ we have the simpler case and we can complete the merge in one pass. In that case we load the first block of R_i for each i from 0 to $N - 1$ into memory and we allocate an output block. Then we choose the first tuple from the all of the blocks, and move it into the output block. If a block R_i becomes empty, replace it with the next block in that run (if there is one). If the output block becomes full, write it out and allocate a new output block. Continue this algorithm until all runs are empty.

If N is large enough that we cannot do it all in a single pass, we will do multiple passes. We will combine the first $M - 1$ runs into a temporary file, and then the next (up to) $M - 1$ runs, and so on, until the last run has been processed. Then we repeat the process using the larger runs as input until we produce the sorted file.

An example may help to clarify. Suppose the file consists of 10 000 blocks and we can fit 50 blocks in memory to do the sort. We can therefore create runs of 10 000/50 = 200 blocks each. Each run is sorted. Then we come to merge them. We cannot fit a block from each of the 200 runs inside the 49 available (50 - 1 block for output) so we must do multiple passes. The first pass sorts runs 1 through 49 into a new run (let's call it R'_1), then runs 50 - 98 into R'_2, et cetera, until the last one which is then 197-200 in R'_5. These larger R' runs are then combined using the same merge procedure as before. Since there are only 5 we are sure the merge will complete in this second pass and we have the output file we wanted.

A visual representation of a two step sort merge from [SKS11]:



Sorting has a cost, of course and the cost analysis also follows from [SKS11]. If the number of blocks in the relation r is b_r, the first step of the algorithm requires us to read in each block of the relation and write it out again for a cost of $2b_r$. The number of runs decreases by $M-1$ in each merge pass, so the number of merge passes is $\lceil log_{M-1}(b_r/M) \rceil$. Each pass reads reads every block of the relation and writes it once, except the last pass doesn't need to write the last output to disk as it is just going to send it on elsewhere. So the total number of block transfers is $b_r(2\lceil log_{M-1}(b_r/M) \rceil + 1)$.

The total number of disk seeks is more complicated: If each run has b_b blocks, then each merge pass requires $\lceil b_r/b_b \rceil$ seeks to read data. And we need to then write the output which costs $2\lceil b_r/b_b \rceil$ for every merge pass, except the last, since it gets sent on. So the total number of seeks is $2\lceil b_r/M \rceil + \lceil b_r/b_b \rceil (2\lceil log_{M-1}(b_r/M) \rceil - 1)$.

What a fun equation that was! Referencing the two sep merge above, we get a total of 44 seeks: computed as $8 + 12 \times (2 \times 2 - 1)$ seeks if the value of b_b is set at 1.

Join Operation

There are several ways to compute a join, and we will analyze the costs of a few of the different approaches. Whatever route is actually taken, it's important to note that joins can be very expensive depending on how this is to be computed. The join we are looking to examine is the one introduced in relational algebra as $\bowtie_\theta$, specifically $r_1 \bowtie_{r_1.a=r_2.b} r_2$ (or in SQL, an `INNER JOIN` with an explicit `ON` predicate). As before, the strategies described come from [SKS11].

Nested-Loop Joins

The simple way to do a join is using one of the nested-loop strategies. It's called the nested loop strategy because that's exactly what it is: we have two nested for loops.

Join Strategy 1: Nested-Loop As with the select operation, the first option we are going to look at is basically a linear search and most likely has terrible performance characteristics. It is, however, a fallback option that we can always choose even if we can't choose anything else. This requires no index exist on either field a or b (the join attributes).

The algorithm described in pseudocode is:

```
1. for each tuple i in r1
2.   for each tuple j in r2
3.     if i.a equals j.b
4.       add (i join j) to the result
5.     end if
6.   end for
7. end for
```

This is, after all, the brute force algorithm, so we expect that the number of tuples that needs to be examined is the number of tuples in r_1 (let's call it n_1) multiplied by the number of tuples in r_2 (n_2). In the best case scenario, both relations fit in memory at the same time and the cost is just two seeks plus a transfer for each of the blocks of r_1 (which we will call b_1) and a transfer for each of the blocks of r_2 (b_2). In the worst case we get only one block of each relation at a time. The inner loop then is one seek plus b_2 transfers to read it into memory. The outer loop requires one seek and one transfer for each block of r_1, plus one run of the inner loop for each tuple in b_1. So the total amount of seeks is $b_1 + n_1$ and the number of blocks to be transferred is $b_1 + n_1 \times b_2$.

To illustrate with an example the difference between the best and worst case scenario, let us assign some numbers to n_1, b_1 and b_2. Suppose n_1 is 1600, b_1 is 100 blocks, n_2 is 10000, and b_2 is 500. Sticking with values used earlier, a seek takes 4 ms and a transfer 0.1 ms. In the best case scenario, the cost is 2 seeks plus a transfer for each block, or: $2 \times t_s + (b_1 + b_2) \times t_T = 2 \times 4 + (100 + 500) \times 0.1 = 8 + 60 = 68$ ms. In the worst case scenario, the number of seeks is $2b_1$ plus the number of transfers as $b_1 + n_1 \times b_2$, or: $2 \times b_1 \times t_s + (b_1 + n_1 \times b_2) \times t_T = 2 \times 100 \times 4 + (100 + 1600 \times 500) \times 0.1 = 200 \times 4 + 800000 \times 0.1 = 800 + 80000 = 80800$ ms. Yikes. That is a really huge difference. Conclusion: buy more RAM.

Assuming that b_1 is not equal to b_2 we can actually observe something interesting: it could be worse! The so-called worst case scenario is actually the second-worst case scenario. Imagine if the order of the relations is switched in the join loop). Let's do the math again: $2 \times b_2 \times t_s + (b_2 + n_2 \times b_1 \times t_T = 2 \times 500 \times 4 + (500 + 10000 \times 100) \times 0.1 = 1000 \times 4 + 1000500 \times 0.1 = 4000 + 100050 = 104050$ ms. So much worse. That's about 1.29 times more just based on the choice of which relation is b_1 and which is b_2.

Under this strategy it matters quite a lot which relation is the outer loop and which one is the inner loop. It would be vastly preferable to make the outer loop the smaller of the two relations since it minimizes the number of seeks as well as reduces the number of transfers.

Join Strategy 2: Block Nested-Loop The previous strategy could be improved by thinking in terms of block operations rather than in terms of individual tuples. Instead of running the inner loop once for each tuple in the outer loop, we could run it once for each block in the other relation and that might be a lot less painful in terms of count of disk reads. Again, we are still assuming that no index is available, so we have to do this the "hard" way.

If everything fits in memory, then the block nested-loop strategy is really no improvement over the regular nested loop algorithm. It gains an advantage if there will be more memory reads. The primary improvement is that instead of loading the inner blocks once for every tuple of the outer relation, they are loaded once per block of the outer relation. Thus there are $b_1 \times b_2 + b_1$ transfers and $2 \times b_1$ seeks that take place.

To once again use the same values as before, there are now $2 \times b_1 = 2 \times 100$ seeks which at a cost of 4 ms = $200 \times 4 = 800$ ms seek time, plus $b_1 \times b_2 + b_1 = 100 \times 500 + 100 = 50100$ at a cost of 0.1 ms = 5010 ms. Summed up a total of 5810 ms to complete this.

As before it could be worse if we swapped the order of the relations. $2 \times b_2 = 2 \times 500$ seeks which at a cost of 4 ms = $1000 \times 4 = 4000$ ms seek time, plus $b_2 \times b_1 + b_2 = 500 \times 100 + 500 = 50500$ at a cost of 0.1 ms = 5050 ms. The total is then 9050 ms which is about 1.56 times the cost of the other ordering.

Of course, you may not lose sleep over this considering that the slower approach here is only 11% of the cost of even the faster simple nested-loop approach. But why not choose the strategy that is 7%?

Join Strategy 3: Index Nested-Loop Suppose an index is available on one of the two relations. If so, then we don't have to do this in the most painful way; we can use the index instead. Since we will iterate over the inner

relation more times, we want the index to be on the inner relation, if we have only one index to work with.

To find which tuples in the inner relation match the outer relation tuple $t_1.a$ then we do an index lookup on r_2 to find the tuple(s) that match. The worst case scenario will once again be that we can only get one block of the relation into memory. We will need to do one seek for each block of the outer relation b_1 and then one transfer for it. We will also need to, once for each tuple in b_1 look into the relation r_2. The formula is $b_1(t_s + t_T) + n_1 \times c$ where c is the cost of doing a single selection in that table using an index (if available). We already know everything we need to know about how to estimate c since we already discussed four strategies for how to do a selection with an equality condition.

As before, the run time is faster if n_1 is the smaller value; that is, the outer relation should have fewer tuples.

Merge Join

The merge join might be more properly called the *sort merge join* approach. Suppose relations r_1 and r_2 have some set of common attributes $r_1 \cap r_2$ and we wish to compute the natural join. If both relations are sorted on $r_1 \cap r_2$ then we can do a merge-sort-like algorithm to compute the join.

Join Strategy 4: Merge-Join A description of the algorithm in pseudocode is in [SKS11] but it it sensible to approach the concepts before looking at what is a long algorithm. The algorithm has one pointer for each relation, initialized to the first tuple in that relation. Then advance those pointers. Then each tuple of r_1 with the same value of the join attributes is read into a temporary block. Then the tuples of r_2 are read and processed as they go.

The aforementioned temporary block does need to be big enough to contain all the tuples of r_1 for each value of the join attributes. That is usually not a problem since we hope not too many tuples match that particular condition. If they do, we would expect either that (1) the database server has a lot of data and therefore needs a lot of RAM and can have the space to do this; or (2) the index is on an inappropriate field (e.g., boolean value) and we should not be trying to use this algorithm. The fallback plan is then the block nested-loop strategy.

Once again, the best case scenario is the two seeks plus reading in each block of each relation. In the worst case, we seek once for each block of the relation as well as a transfer for each block of each relation: $(b_1 + b_2)(t_s + t_T)$.

There is also an assumption that the relations in question are sorted on the attributes of the join. While that may be true if one of them is a primary key attribute, it is probably not the case for the other relation. As you would imagine, a join of employees and departments relies on relations employee and department and it is unlikely that employees will be sorted based on department ID or vice versa. Each will have their own primary key.

Join Strategy 5: Hybrid Merge-Join We can execute a variant of the merge-join approach on unsorted tuples if a secondary index exists on the join attribute(s). Normally if we are accessing the second relation through an index of some sort, we get the tuples in sorted order (from the view of that index), but they could be anywhere in the disk blocks meaning that each time we go to the next tuple is potentially another disk seek and transfer.

To save ourselves some work we can use the hybrid merge-join algorithm. The basic idea here is to merge the sorted relation r_1 with the leaf entries of r_2's secondary index. The output is then made up of tuples of r_1 and pointers to the tuples of r_2. This file can then be sorted on the pointers to r_2 tuples so that accesses to r_2 go in physical storage order rather than jumping everywhere back and forth.

Join Strategy 6: Hash Join Another important strategy can be used for joining relations and it relies on hashing as we have already discussed. However, the complexity of the hash join is high and we will consider it beyond the scope of this course. Those who are curious can find more information in [SKS11].

Other Considerations in Query Processing

There are a few more situations beyond the select and join scenarios that we will care about when we have to actually execute a query. Let us see a few more things from [SKS11] that note some extra work we may have to do if the query is slightly more complex.

Distinct. Often times in performing a query we are asked to remove duplicates (e.g., select distinct). This is easy enough if we are sorting the data because two identical values will be next to each other for merging. Alternatively, we could simply avoid writing them to the output at all if the sort-inserted routine finds something already in the designated space for that value. This can be expensive, so this is generally done only if we really ask for it.

Projection. Projection should probably get a mention: it is pretty much just throwing away any columns of the relation that we don't need in some way. Projection could happen in several steps; if employee ID is used in a where condition, for example, but not asked for in the output, we might first project the full tuple down to the output fields plus those used in the where condition, then later throw away some more when presenting the final results. The same is true of an order-by clause: if that field is not needed in the output we will toss it out once the ordering is complete.

Set Operations. Set operations (union, intersection, difference) are fairly self explanatory as well. Sorting the data is necessary, generally, to make this efficient. If there is no index, we might need to build one. Each relation r_x is partitioned into different parts which we will label r_{x_0}, r_{x_1}.... A quick recap from [SKS11] on how to do the various operations:

- Union
  1. Build an in memory hash index on the r_{1_i}
  2. Add the tuples r_{2_i} to the index if they are not present
  3. Add the tuples referenced in the index to the result.
- Intersection
  1. Build an in memory hash index on the r_{1_i}
  2. For each of the tuples in r_{2_i}, see if it is already in the index, and if so, add it to the output.
- Difference
  1. Build an in memory hash index on the r_{1_i}
  2. For each of the tuples in r_{2_i}, see if it is already in the index, delete it from the index.
  3. Add the tuples referenced in the index to the result.

Outer Join. The outer join operation requires us to add some tuples that don't have a match in the other relation. A nested loop algorithm isn't too bad: if there is no match then pad the tuples with nulls on the appropriate side(s) and add it to the output. An alternative strategy is then to compute the (regular) join, then do a select on all tuples, subtracting those without a match, and pad those tuples and add them to the result.

Aggregation. One approach to aggregation is more or less the same as the idea of duplicate removal. We select all the ones we need to aggregate, then take a second pass through the data to sort it, and then merge the values as are necessary. But this is probably not the best way, as might occur to you when thinking about count. Count might be done slightly more efficiently since we can skip the sort and merge entries part by simply keeping a running count. That might also be applied to min, max, sum if we are clever enough to maintain those values and update them accordingly. Average would require maintaining two values and then performing a division... but somehow this seems rather possible.

Order of Operations. When dealing with simple expressions there is one operator and we have seen that it can sometimes matter in which order the operands occur. This problem of order of operations becomes larger when we start having complex queries. $r_1 \bowtie r_2 \bowtie r_3$ could be executed with two different groupings and with different orders of operands:

1. $(r_1 \bowtie r_2) \bowtie r_3$,
2. $(r_2 \bowtie r_1) \bowtie r_3$,
3. $r_3 \bowtie (r_1 \bowtie r_2)$,

4. $r_3 \bowtie (r_2 \bowtie r_1)$,
5. $r_1 \bowtie (r_2 \bowtie r_3)$.
6. $r_1 \bowtie (r_3 \bowtie r_2)$.
7. $(r_2 \bowtie r_3) \bowtie r_1$
8. $(r_3 \bowtie r_2) \bowtie r_1$

Sometimes we do not have quite this degree of freedom, however, and an order is necessarily implied, such as a subquery. In the next topics we will examine evaluations of expressions and look in more detail at how we decide which approach is likely to be best.

18 — Query Evaluation and Optimization

Query Evaluation

The strategies we have looked at thus far have explained how to perform individual parts of a query. If a simple select statement or a select-join is needed then we just have to carry it out using an evaluation plan. Choosing which evaluation plan, as we saw, is not trivial, and soon we will examine how we can "guess" about which plan is likely to be best. But first we need to think about what order in which to perform a compound expression.

Evaluation of Expressions

Our normal expectation of how a composite operation works is that we do a subpart (whichever one) and then store the resulting relation temporarily for further use. This is called *materialization* and usually results in the temporary relation being written to disk. That seems undesirable but it may be necessary. The other option is a *pipeline* which would allow us to immediately forward on the partial results from a particular operation so the second operation can run in parallel with the first [SKS11].

Materialization. First, let us talk about materialization. Sadly, this is not about how people who "beam up" using the transporter in Star Trek get put back together again afterwards. In the example from earlier where there is a three way join $r_1 \bowtie r_2 \bowtie r_3$ we will choose one of the ways to group this and then, as you would expect, temporarily store the output in some temporary relation r_4 which is then used in the join with the third relation.

Calling back to an idea you have likely studied in compilers (or will study soon at least), when presented with a complex statement, we need to parse it and form a tree. Then, we work from the bottom level of the tree up to complete the expression.

If the code to be compiled is `x = (y * 5) + z;` we can look at that and discover we need to do the following things: fetch y, multiply y by 5 and store it in a temporary location, fetch z, add y to z, and finally, assign it to x. There are some things we can do in a different order, potentially, such as fetching z in a different order, but there are some operations we can't do out of order like addition before the brackets[12].

In the database, we again, need to work on our low level operations and execute those operations (fetch rows, compute joins, whatever it is) and take the output and put this in a temporary relation. A temporary relation will be created at each step until all operations are complete. At that point we have the final result and can return it.

The cost of materialization is the sum of the individual operations, plus the cost of writing all intermediate steps to disk [SKS11]. How large those intermediate costs are depends very heavily on how much data is to be written to disk at each step. However many tuples of each intermediate step fit into a block is important because it determines how many blocks are needed at each step. Once again, we get a hint that says if we get some choices about which operations to do sooner rather than later, we want the ones that result in the fewest result tuples in the output relation...

Pipelining. The idea behind pipelining is to reduce or avoid the costs of storing those temporary files. The idea of pipelining is very common in computing, from CPUs all the way up to large software transactional systems.

[12] If anyone ever tries to tell me that order of operations doesn't matter much, I usually tell them to make a cake by first baking all the ingredients and then mixing them.

In short, in pipelining, a chunk of data moves through several different stages from beginning to end. When a particular chunk moves from stage 1 to stage 2, the next chunk can move into stage 1 and begin processing. Thus, we can accomplish things in parallel and potentially get some more done in less time.

Without pipelining, the first data item to finish stage 1 just kind of sits around waiting for the last item to finish stage 1 before the first item can start stage 2. It accordingly needs a place to wait. Eliminating that place of waiting is the goal of pipelining. And this is the plan for execution. As a nice bonus, we can maybe show partial results to the user as processing continues rather than waiting until all is done before showing all results.

In [SKS11] there are two approaches for how pipelines may execute, either demand-driven (or consumer-driven, pull) pipeline, the system requests at the output end the "next" chunk when it is ready to receive it, which triggers the earlier stages in turn to pick up their next chunk of data and so on and so on. In the supply-driven (or producer-driven, push) pipeline, each stage of the pipeline is always trying to pick up the next chunk and process it (but it may be blocked waiting for space to open up in the output area).

My personal preference runs towards a push-driven pipeline since it rather resembles the type of producer-consumer problem discussed in earlier courses. Each stage of the pipeline can be its own thread, and threads can be both a producer and a consumer. The first stage reads the first chunk of the input file and does some transformation, and places that in its output buffer. Obviously, some sort of mutual exclusion constructs such as semaphores will be used to get proper concurrency control here. The second stage can then pick up that intermediate data, process it, and pass it on to the third stage[13] by putting it in yet another intermediate buffer. This continues until all data is processed through all stages and is output. Buffer sizes will be limited, of course, so there can be different stages that are blocked awaiting space in a buffer or waiting their turn to access a buffer. But this sort of program structure should be quite familiar to you from learning about semaphores and the like.

The catch, when it comes to pipelining, is that there are some operations that don't work very well with it. Sorting is the most obvious example: you cannot send the first chunk of the sorted file on to the next stage until the entire file has been sorted. Our choice of evaluation algorithm for a select or join operation may also limit the ability to pipeline. These sorts of things are just limitations we cannot easily get around and may need to be accounted for in our plan for how to evaluate a query.

Query Optimization

Finally we are ready to talk about query optimization. This topic should be pretty easy to motivate. The database server is responsible for choosing how to carry out the requested query and it is preferable to do it efficiently and choose an optimal plan rather than a sub-optimal one. When we put some numbers to early examples, we saw that there can be orders of magnitude difference in how long it takes to execute a query depending on the strategy, and even which operand is on the left or right side of the operator. Conclusion: it is worth while for the database server to make the effort to decide what is optimal.

We have looked at techniques for evaluation of how long we think a certain plan is going to take to execute which concentrated primarily on how many disk operations we expected to take place. These typically included numbers like the number of tuples or the number of blocks and that left open the question of how did we know how many tuples or blocks are in this relation?

One way that might have immediately come to mind is the metadata kept by the database system; if it does provide us with some useful information like relation r has 10592 tuples and is currently stored in 2591 blocks then we have some values to go on. Those are the easy values to get. But how do we determine how many tuples we think will match the condition that some attribute A is greater than some value x? And how many blocks will those tuples be in?

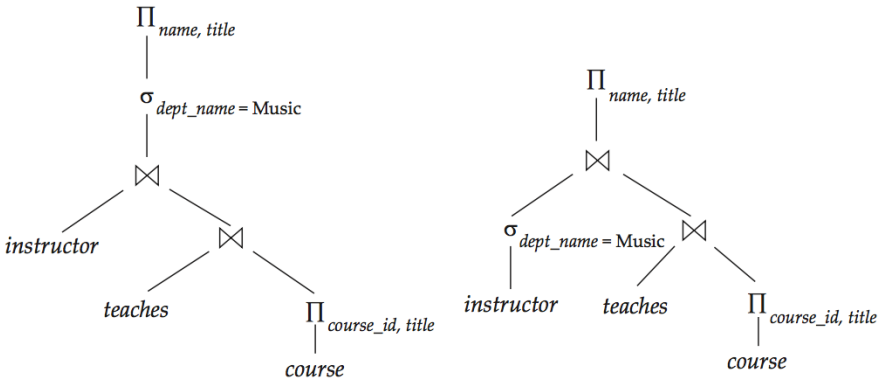
The answer lies mostly in *heuristics*, or perhaps less charitably, educated guessing. A heuristic is a guideline or "rule of thumb" the gives us some hints. Suppose we know some statistics about A. If we do, then it can help us make some important decisions about what execution plans we choose.

In fact, one of the main heuristic rules we should always try to follow is to cut down the size of intermediate relations whenever possible. That means select and project operations should be done early and certainly before

[13] `https://www.youtube.com/watch?v=KzHX_MP_eBk`

a join. A join operation can result in a file size that is some multiple of the size of the input file so it is preferable to cut things down first [EN11].

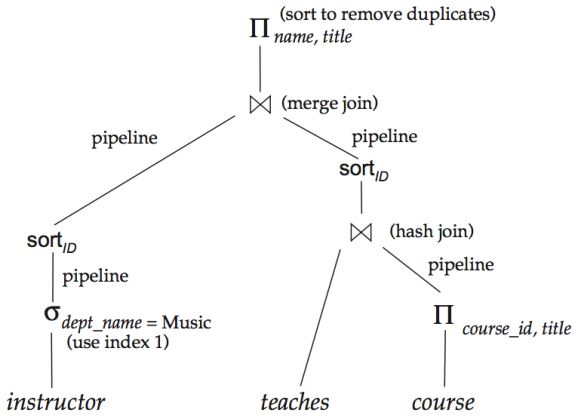
We previously introduced the idea of turning the input query into a tree structure which was then evaluated from the bottom up. Combining that with our knowledge that order of operations matters and that we would like to choose the efficient route, we will often transform the input expression to a faster, better equivalent. See the diagram below [SKS11]:



(a) Initial expression tree (b) Transformed expression tree

If it seems strange to you that the expressions can be rewritten a bit, it is more normal than one might expect. Compilers do this all the time, generating code that is equivalent to, but much faster than, the code that you wrote in the source file. If you would like to hear more about that, you could take ECE 459, Programming for Performance!

This is only one possible transformation. Depending on the nature of the query, zero, one, or multiple equivalents may exist. It may be practical to write off some of those immediately, without assessing them, because we know they will be an inferior variant of a plan we have. The next step is then to draw up the plans for how we would execute the expressions. An annotated plan looks something like this [SKS11]:



There may be many options for annotating each of the steps in the expression tree. A totally thorough approach would, once again, consider every possibility, but we realistically can eliminate some options that we know will be inferior to other choices. Then we may affix cost estimates to each plan as the sum of each part. The estimates can be wrong (it's why they are called estimates) but generally we can be sure we are at least in the correct ballpark. The final step is then to choose the the plan with the lowest cost. That last step is probably the least interesting since it just requires us to, well, compare some numbers and choose the smallest one.

As a practical note, in SQL you may ask the database server to tell you how it would carry out a query. First you write the query as you normally would and prefix this query with the keyword `EXPLAIN`[14].

[14]Every time I do this, I can't help but feel like this: `http://imgur.com/lUIlU4l`. I think the all-caps nature of the keyword contributes.

Expression Transformation

There are some rules for how exactly to transform a query to an equivalent alternative. These are a little bit like some things you may have learned in mathematics, like how multiplication commutes or that addition is associative or something like that. There is probably nothing really saved by re-ordering a simple math expression like $12 \times 2 \times 3$, although you may find it personally faster or easier to do the math in your head by first computing 2×3 and multiplying the result by 12 rather than first computing 12×2 and then multiplying it by 3. Or you may personally find it easier to do the opposite, or have no preference whatsoever. But the idea applies.

We should remember that database operations do not, unless there is an explicit order-by clause, specify an order in which tuples appear in the output. Order does not matter in the output. If a request is for addresses where the province is "ON" or the province is "QC" then one execution plan may result in all tuples where province is "ON" followed by all provinces where the province is "QC" or the reverse, or they may be all mixed together. All three of those outcomes are equivalent as long as the same tuples are in the output.

There are some rules from [SKS11] and [EN11] that we want to learn about. To spare a lot of repetition of what things mean, the notation will be centralized here. θ_x denotes a predicate (as part of a selection or join, for example), L_x denotes lists of attributes, and E denotes a relational algebra expression (a sub-expression or a relation r). All our other symbols from relational algebra remain the same as when they were first introduced.

Rule 1: Conjunctive Selection Conjunctive selection can be turned into a sequence of individual selections. This makes sense: if we have a selection from address where province is "ON" and city is "Kitchener", we can do this as a selection on address where city is "Kitchener", producing a temporary relation and then we do a selection where province is "ON" on that temporary relation. That is sometimes called a cascade of selection.

In relational algebra: $\sigma_{\theta_1 \wedge \theta_2}(E) = \sigma_{\theta_1}(\sigma_{\theta_2}(E))$

Rule 2: Selection Commutes Selection commutes. Suppose the original selection is on address where city is "Kitchener", producing a temporary relation and then we do a selection where province is "ON" on that temporary relation. This is equivalent to doing a selection on address where province is "ON" and producing a temporary relation, and then we can select where city is "Kitchener" on the temporary relation. The results are identical in both cases.

In relational algebra: $\sigma_{\theta_1}(\sigma_{\theta_2}(E)) = \sigma_{\theta_2}(\sigma_{\theta_1}(E))$

Rule 3: Projection Redundancy Elimination Only the last projection operation is needed; all other projection operations do not do anything. This should be pretty logical given that projection reduces the returned attributes to just those that are specified in the projection.

In relational algebra: $\Pi_{L_1}(\Pi_{L_2}(\Pi_{L_3}(E))) = \Pi_{L_1}(E)$

Rule 4: Commuting Selection and Projection If a selection involves the same attributes as a projection list, we can commute the two operations. That is, if L contains exactly the same set of attributes $A_1, A_2...$ that are referenced in θ then:

$\Pi_L(\sigma_\theta(E)) = \sigma_\theta(\Pi_L(E))$

Rule 5: Selection Combination Selection may be combined with both cartesian product. This is, as you will recall, how a theta join works and is shown as (a) below in the relational algebra. If a selection being combined with what is already a theta join, that is the same as a theta join with a conjunctive predicate, shown as (b) below in relational algebra.

a. $\sigma_\theta(E_1 \times E_2) = E_1 \bowtie_\theta E_2$

b. $\sigma_{\theta_1}(E_1 \bowtie_{\theta_2} E_2) = E_1 \bowtie_{\theta_1 \wedge \theta_2} E_2$

Rule 6: Theta Joins Commute Theta (and natural) joins commute, so the operands can be swapped in order and the same result is produced. That is a relief, considering our previous discussion of what blocks need to get

moved into memory to complete the operation. This does potentially change the order of the attributes in the output, but we should not actually care. Normally a selection statement, for example, specifies some attributes to be returned, meaning there is a projection operation to be done. If there is a select $*$ operation, no order is guaranteed in the output anyway.

In relational algebra: $E_1 \bowtie_\theta E_2 = E_2 \bowtie_\theta E_1$

The natural join is a special case of the theta join, so $E_1 \bowtie E_2 = E_2 \bowtie E_1$

Rule 7: Natural Join Associates When discussing joins, we actually already covered this rule, that they are associative, i.e. we can do them pairwise in whichever order we prefer. For theta joins, it looks a little more complicated, but the outcome is the same, keeping in mind that θ_2 references only attributes of Expressions 2 and 3 (so it can't really be applied to Expression 1).

a. $(E_1 \bowtie E_2) \bowtie E_3 = E_1 \bowtie (E_2 \bowtie E_3)$.

b. $(E_1 \bowtie_{\theta_1} E_2) \bowtie_{\theta_2 \wedge \theta_3} E_3 = E_1 \bowtie_{\theta_1 \wedge \theta_3} (E_2 \bowtie_{\theta_2} E_3)$.

c. Because cartesian product is an empty theta join, $(E_1 \times E_2) \times E_3 = E_1 \times (E_2 \times E_3)$.

Rule 8: Selection Distribution Selection distributes over theta join if the following conditions hold:

a. It distributes if all the attributes in θ_0 involve only the attributes of one of the expressions E being joined. This means we can cut down the first relation to only the matching rows before performing the join, which is hopefully a faster way to do it. In this example, if θ_0 applies only to E_1:

$\sigma_{\theta_0}(E_1 \bowtie_\theta E_2) = (\sigma_{\theta_0}(E_1)) \bowtie_\theta E_2$

b. It distributes when θ_1 involves only the attributes of E_1 and θ_2 only the attributes of E_2. This would mean cutting down both relations before performing the join.

$\sigma_{\theta_1 \wedge \theta_2}(E_1 \bowtie_\theta E_2) = (\sigma_{\theta_1}(E_1)) \bowtie_\theta (\sigma_{\theta_2}(E_2))$

Both of these scenarios apply also for the cartesian product.

Rule 9: Projection Distribution The projection operation can be distributed over theta join if the following conditions hold:

a. If L_1 and L_2 are attributes of E_1 and E_2 respectively, and θ contains only attributes in $L_1 \cup L_2$, then we can distribute. This means that we can, much like the previous rule, cut down the relations using the projection before performing the join:

$\Pi_{L_1 \cup L_2}(E_1 \bowtie_\theta E_2) = (\Pi_{L_1}(E_1)) \bowtie_\theta (\Pi_{L_2}(E_2))$

b. If L_1 and L_2 are attributes of E_1 and E_2 respectively, and L_3 are join attributes of E_1 not $L_1 \cup L_2$ and L_4 are join attributes of E_2 not $L_1 \cup L_2$:

$\Pi_{L_1 \cup L_2}(E_1 \bowtie_\theta E_2) = \Pi_{L_1 \cup L_2}((\Pi_{L_1 \cup L_3}(E1)) \bowtie_\theta (\Pi_{L_2 \cup L_4}(E_2)))$

Again, this can also be applied to cartesian product.

Rule 10: Set Operations Commute The set operations union and intersection commute (difference does not).

a. $E_1 \cup E_2 = E_2 \cup E_1$

b. $E_1 \cap E_2 = E_2 \cap E_1$

Rule 11: Set Operations Associate Union and intersection are associative:

a. $(E_1 \cup E_2) \cup E_3 = E_1 \cup (E_2 \cup E_3)$

b. $(E_1 \cap E_2) \cap E_3 = E_1 \cap (E_2 \cap E_3)$

Rule 12: Selection Distribution II Selection distributes over union, intersection, and difference operations:

a. $\sigma_\theta(E_1 \cup E_2) = \sigma_\theta(E_1) \cup \sigma_\theta(E_2)$

b. $\sigma_\theta(E_1 \cap E_2) = \sigma_\theta(E_1) \cap \sigma_\theta(E_2)$

c. $\sigma_\theta(E_1 - E_2) = \sigma_\theta(E_1) - \sigma_\theta(E_2)$

Rule 13: Projection Distribution II The projection operation distributes over a union operation.

In relational algebra: $\Pi_L(E_1 \cup E_2) = (\Pi_L(E_1)) \cup (\Pi_L(E_2))$

Rules Review

These rules are (sadly) neither a complete set nor minimal. There are other potential transformations that can be done, including various rules from boolean algebra (e.g., $\neg(a \wedge b) = \neg a \vee \neg b$) that are not covered here [EN11]. As we might see from looking at the rules as well, it is possible to use certain rules to derive other ones in that list. This is okay, as they are intended as a human comprehensible list. If we were actually writing these for the computer we would probably want a minimal set to reduce the amount of computation done to generate all the equivalent transformations.

Having learned a bit about equivalence rules and what they are for, next time we will work on putting them to use and affixing cost estimates.

19 — Query Optimization

Query Optimization

Given an understanding of the various rules, there are now some clear ways we can transform an input query to determine which of the equivalent representations will have the lowest cost of execution. Combining that with what we have already seen when it comes to join operations, it is fair to say that the query optimizer is likely to focus first on join relations since that is potentially the biggest area in which we can make some gains.

Suppose our query involves a selection and a join: we want to select the employee number, salary, and address for an employee with an ID of 385. Suppose number and salary are in the employee table with 300 entries, and the address information is in another table with 12000 entries. We have a join query, and if we do this badly, we will compute the join of employees and addresses, producing some 300 results, and then we need to do a selection and a projection on that intermediate relation. If done efficiently, we will do the selection and projection first, meaning the join needs to match exactly one tuple of employees rather than all 300.

The query optimizer should systematically generate equivalent expressions, but since performing all possible transformations and then evaluating each option may itself take non-trivial time, it is likely that the optimizer does not consider every possibility and will take some "shortcuts" rather than brute force this. One technique that helps on top of that is to re-use common subexpressions to reduce the amount of space used by representing the expressions during evaluation [SKS11].

Estimating Statistics

For all the discussion about how it might make sense to swap this or change that or do this rather than that, we have not yet really talked about how the system may guess about how many results are going to be returned. In the previous example I used exact numbers, 300... 1... 12000... etc., but for the database server to get those it can either look them up, or it can guess about them. As mentioned earlier, sometimes certain numbers, like the number of tuples in a relation, are easily available by looking at metadata. If we want to know, however, how many employees have a salary between $40 000 and $50 000, the only way to be sure[15] is to actually do the query, and we most certainly do not want to do the query when estimating the cost, since at that point we might as well not bother optimizing at all.

If we cannot measure, then, well, we need to guess. Our guesses are, however, not wild, but instead educated. Estimates are based on assumptions, and those assumptions are very often wrong. That is okay. We do not need to be perfect. All we need is to be better than not optimizing. And even if we pick the second or third or fifth best option, that is acceptable as long as we are close to the best option.

There are five major areas where costs for actually performing a query accumulates. We have, for simplicity, acted as if accessing disk storage is much larger than any other costs to the point where it dominates all other factors. That is not quite accurate, and modern query optimizers are capable of taking more things into consideration. The list from [EN11] below is more complete:

1. **Disk I/O**: the previously-discussed costs of seeking and data transfer to get data from the disk or other secondary storage.

[15] Other than nuking it from orbit...

2. **Disk Additional Storage**: the costs, if any, of storing intermediate relations on disk during processing of the query.
3. **Computation**: The CPU usage that will be required to do the operations including sorting, merging, and arithmetic operations.
4. **Memory**: The memory usage for the buffers necessary to do the desired operations.
5. **Communication**: The cost of sending the results back to the requester, or in a distributed database, the cost of transferring data between the different nodes.

We will generally proceed on the basis that disk I/O is the largest cost and outweighs everything else. That may not be true in a particular use case (e.g., the database server has tons of RAM and a lot of data can be kept in memory). The principles of assessing cost and choosing a strategy still apply, even if we are optimizing faster queries or constrained by CPU instead of disk.

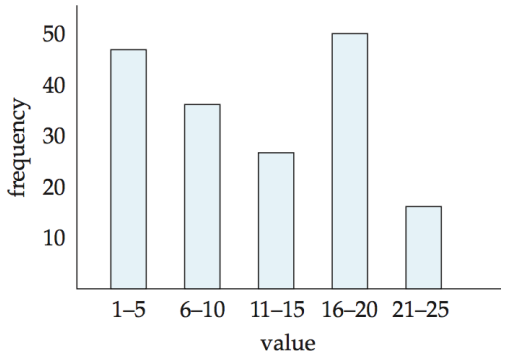
Metadata. As previously mentioned, there is some metadata stored in the database that we could look at to find out some data that we know has some high accuracy. Some items that might be in the metadata, from [SKS11] and [EN11]:

- n_r: the number of tuples in a relation r
- b_r: The number of blocks containing a relation r
- l_r: the size in bytes of relation r
- f_r: the number of tuples of r that fit into one block
- $V(A, r)$: the number of distinct values in r of attribute A
- $h_{r,i}$: the height of an index i defined on relation r

Some of these values can be computed, notably l_r is the number of blocks times the size of a block, and f_r is the number of tuples divided by the number of blocks. The value of $V(A, r)$ may or may not be maintained for all attributes, or for groups if that is so desired. If it is on a key field, every value is unique, so we know it is the same as the number of tuples for that field. There can also be metadata about index information as well... which might make it metametadata?

There are obviously tradeoffs when it comes to maintaining metadata. The more often it is updated, the more effort is spent updating it. If every insertion or update or deletion resulted in an update, that may mean a nontrivial amount of time is spent updating this data. If we only do periodic updates, it likely means that the statistic data will be outdated when we go to retrieve it for use in a query optimization context. Perhaps some amount of balance is necessary: updates can be done during times of lower system load with some maximum limit for how long an update can wait before it gets enough priority. This is, in some small way, a callback to scheduling algorithms you have surely learned about in an OS class.

A database may also be interested in keeping some statistical information in a histogram. The values are divided into ranges and we have some idea of how many tuples are in those ranges. You have almost certainly seen a histogram before in something like a population pyramid diagram. An example from [SKS11]:



That should also tell you that they do not necessarily have an even distribution. A histogram does not take up a lot of space and it can help us to figure out certain problems: if we have a histogram of salaries and the largest bucket is 100 000+ and there are 10 people in this category, and there is a query asking how many employees have a salary greater than or equal to $100 000 we will know at least that the number of tuples to be returned is 10. Nice.

The above numbers are exact values which we can know and, hopefully, trust although they could be slightly out of date depending on when exactly metadata updates are performed. The more exact values we have, the better our guesses. But things start to get interesting when, in the previous example, we ask something that does not have a category, such as how many people have a salary larger than $150 000, where there isn't an obvious answer found in the metadata?

As in the previous discussion of costs, we expect select operations and join operations to be the important ones since they are the ones we will do the most of (probably). Our strategies again come from [SKS11]:

Estimating Selection Size

We will start with select operations.

Single Equality Predicate. The base case to consider is a single predicate in a selection with an equality condition (e.g., province = "ON"). The estimate depends on what we know about the attribute A referenced in the selection. If it is a key field, and therefore unique, we know the select will return 1 result. Unless, of course, it is not found and we will get zero results; but usually we do not expect that sort of result.

If values are evenly distributed, then we can expect that $n_r/V(A,r)$ tuples should be returned. If there are ten possible values and everything is approximately evenly distributed then a select is expected to produce approximately 10% of the tuples as its result. The uniform distribution may not be accurate, but if it is a reasonable approximation of the data, it is simple and fast. If a histogram is available, we can look in the range that contains that search value and guess based on the frequency count for that range instead.

Single Comparison Predicate. The next case is when we have a comparison like the salary greater than or equal to $150 000 then we can, using the known maximum and minimum values of this attribute, take a guess. If the requested value v is less than the minimum, we expect no rows returned; if the requested value is greater than the max also zero. Otherwise, the formula to estimate, where m is the minimum value and M is the maximum value: $n_r \times \frac{v-m}{M-m}$

Of course, if a histogram is available, we would prefer to use that.

Complex Predicates. For conjunction as well as disjunction, we will use a new symbol, s, which represents the selectivity of the a particular selection: how likely it is that a particular tuple matches a condition. The probability that a tuple satisfies θ_i is s_i/n_r.

For conjunctive selection, we want the probability that a certain tuple meets all of the criteria. So the formula to estimate is:

$n_r \times \frac{s_1 \times s_2 ... \times s_k}{(n_r)^k}$.

For disjunctive selection it is the probability of matching an one of those conditions. The calculation will be simplified by trying to compute 1 minus the probability it satisfies none of the conditions:

$1 - (1 - \frac{s_1}{n_r}) \times (1 - \frac{s_2}{n_r}) \times ... \times (1 - \frac{s_k}{n_r})$

Negation is pretty simple: if we have a predicate $\neg\theta$ then the number of tuples likely to meet that is the number of tuples in the relation subtract those that do match θ.

Estimating Join Size

If we are computing a join where the relations have no attributes in common, then the "join" is really the cartesian product and the number of tuples in the output will be the number of tuples in r_1 multiplied by the number of tuples in r_2. That's a lot.

If the intersection of r_1 and r_2 is a key (unique value) for r_1 then we know that a tuple in r_2 can match with at most one in r_1 so the maximum number of tuples that could occur is the number of tuples in r_2. If the intersection is also defined as a foreign key on r_2 then it is exactly the number of tuples in r_2. This also applies symmetrically (so you can swap r_1 and r_2 in the previous statements).

The difficult case occurs if the intersection of two relations is not a key in either of those two relations. In that case, looking at the attributes in the relation, which we will call A, then the number of tuples in the join relation are:

$$\frac{n_{r_2}}{V(A, r_2)}.$$

In other words, the average number of values in r_2 that have a given value of A. Since we expect that they match with r_1 then we estimate that there will be:

$$\frac{n_{r_1} \times n_{r_2}}{V(A, r_2)}.$$

Now wait, you might say, because the denominator here depends only on r_2, and if you switched the positions of r_1 and r_2 we could get a different answer if $V(A, r_1)$ is not identical to $V(A, r_2)$. Quite right. Those two values may be different if not all entries in one of the tables can be joined with an entry in the other. That is unusual but not impossible. If that is the case, we would choose the smaller of the two estimates since the smaller relation will limit what we can do.

The previous calculation does assume, of course, that all values are equally likely. If they are not, then applying a histogram will give a more accurate prediction.

Other Operations

Projection. Projection on an attribute A will result in either n_r tuples if duplicates are allowed, or $V(A, r)$ if they are not. The relational algebra definition does not permit duplicates, mind you.

Aggregation. Suppose we seek an aggregation on some attribute B grouping by A. Imagine in an example that B is salary and A is job title. Then there will be one tuple in the output for each distinct value of A.

Set Operations. If a set operation has both operands as the same relation, it can be rewritten as a compound predicate and that would give us our estimates. Those sorts of transformations are not especially difficult to imagine: $\sigma_{\theta_1}(r) \cup \sigma_{\theta_2}(r)$ is easily rewritten as $\sigma_{\theta_1 \vee \theta 2}(r)$.

If the set operation is on different relations then we could actually estimate it directly. We will typically assume worst case scenario, so: (1) union will be the size of the two relations, (2) intersection will be the size of the smaller of the two relations, and (3) in set difference we will assume in $r_1 - r_2$ that the result is the size of r_1 (i.e. no tuples removed).

Outer Join. Outer joins are fairly simple. For a left outer join: $r_1 \mathbin{⟕} r_2$ the estimated size is $r_1 \bowtie r_2$ plus the size of r_1; right outer join $r_1 \mathbin{⟖} r_2$ is estimated as $r_1 \bowtie r_2$ plus the size of r_2 and the full outer join $\mathbin{⟗}$ is just the combined size of r_1 plus r_2.

Admittedly, I beg to differ from the textbook on the size of the outer joins. We know that in a left outer join, for example, there will be r_1 tuples in the output because one tuple will appear for each tuple in r_1 whether it has a match in r_2 or not. The right outer join is symmetric. I would therefore save the headache of trying to estimate (or even reuse) an estimate of $r_1 \bowtie r_2$. By this same logic the full outer join does agree with the previous paragraph.

Distinct Values

If we are interested in knowing how many distinct values of an attribute A there are, we have a few shortcuts based on the query. The simplest possible case is if the query specifies that A equals some value a in which case there will be at most 1 distinct value of A. If the predicate is complex and there are several allowed values $a_1, a_2, a_3 ... a_n$ then we can be certain the maximum number of possible values of A is n. The last easy case is if there is a comparison, in which case the estimate is $V(A, r) \times s$ where s is the selectivity of the selection.

Otherwise we will assume that the distribution of values of A has no relation to the distribution of values in the selection predicate. The correct way to guess is through the use of probability theory, but we could also do a quick estimate of the smallest value of $V(A, r)$ (the most restrictive selection condition).

This same logic holds for estimating how many unique values will be returned in a join: the smallest condition will limit how many rows are in the relation. Whether that is in the left relation or the right depends heavily on the join conditions.

Advanced Query Optimization

There are a few more advanced techniques which we really will not dig into in great detail, but will examine in some degree.

Top-K. Consider a query that asks you to find the top k tuples based on some attribute, such as finding the top ten customers based on annual revenue. We could potentially save some work if we know that no further results are required. Pipelining might be sufficient if the conditions are simple enough that we can perhaps sort and then process the data. But for something like top customers by revenue, we would probably need to find all the tuples that match, aggregate them, and then sort by income. Heuristics might help, but might not: generally more invoices means more income, but we could also have a customer with one invoice for $10 000 000.

Update Ordering. Update queries often have a bunch of queries restricting them to one tuple. If that is the case, update optimization is not really an issue. But if the update affects a number of rows, then there is a possible problem called the *Halloween problem*, apparently because it was discovered on that day. Suppose an update is being done in a pipelined fashion: an updated tuple could be inserted into the relation ahead of where the scan is, meaning the tuple could be updated again incorrectly.

The problem can be solved in a few ways. One is to determine the tuples to be modified first before any updates take place. Another has to do with splitting up the updates to be done into batches so that we don't have to wait for the whole selection to be done.

I would argue that these sorts of update-ordering problems can be more properly solved by good transaction management. But this is a topic we will have to return to.

Multiquery Optimization. The first way we can optimize multiple queries at once is to avoid repeating any operation that is used in more than one query. That is to say, if query 1 and query 2 both have some part in common (e.g., the same subquery), the result of that subquery should be reused rather than recomputed a second time. That would, of course, require that query 1 does not modify the data, which could cause a problem.

The second way to optimize when we have multiple queries is to do some things in parallel. If query 3 scans over a particular relation r looking for some attributes, and query 4 scans over the same relation r looking for something different, then we could note the tuples matching queries 3 and 4 in one single pass rather than having to scan it twice.

20 — Query Optimization and Plan Selection

Query Optimization

The final advanced subject in query optimization is complex enough that it deserves a more detailed look: join elimination. So much of the previous examination has focused on the cost of the join and that has highlighted in a real way just how expensive it is to perform a join. For this reason, good optimizer routines will attempt to eliminate the join altogether if it can be skipped. The optimizer can only do this if there is certainty that the outcome will not be affected by not doing the join. We will shortly see how that is accomplished.

Join Elimination

You may ask, of course, why should the optimizer do this work at all? Why not simply count on the developers who wrote the SQL in the first place to refactor/change it so that it is no longer so inefficient? That would be nice but would you also like a pony? Developers make mistakes, as you know, or perhaps some legacy code cannot be changed for some reason. Regardless, SQL is a language in which you specify the result that you want, not specifically how to get it. If there is a more efficient route, then it's worth taking from the point of view of the database server. The same logic applies in the compiler, incidentally; if you ask for some operation that the compiler knows it can replace with an equivalent but faster operation, why wouldn't you want that? Compilers don't admonish the user for writing code that it has to transform into a faster equivalent, they just do that transparently[16].

Our guide in this is [Ede17a]. We will examine some real SQL queries to see how we can get rid of a total unnecessary join. This join can only be removed if the database server can prove that the join is not needed and that therefore the removal of this operation has no impact on the outcome.

Consider a statement that looks like this: `SELECT c.* FROM customer AS c JOIN address AS a ON c.address_id = a.address_id;` This gets customer information and joins with those where there are addresses on file. This is an inner join and as presented simply we cannot do anything with this information. We need to make sure that the customer data has a matching row.

If, however, a new piece of information is considered, it changes everything entirely. Suppose that we have a foreign key defined from customer's `address_id` to the address `id` field. If nulls are not permitted then we know for sure that every customer has exactly one record in the address table and therefore the join condition is not useful and may be eliminated. This means we could in fact replace that query with `select * from customer;` with no need for any references to the join table at all. That would be much, much faster since it is a simple select with no conditions.

The foreign key and not null constraints on the address ID field of the customer make it possible for the optimization of the join elimination to occur.

An outer join constraint can be removed as well. Imagine the query said this: `SELECT c.* FROM customer AS`

[16] Of course, if you'd like a peek behind the curtain to see what compiler optimizations do, ECE 459, Programming for Performance, does examine some parts of this!

`c LEFT OUTER JOIN address AS a ON c.address_id = a.address_id;`. All tuples are fetched from customer whether or not there is an associated address. Once again, if the foreign key constraint and not-null properties hold for this field it means that we can perform the equivalent transformation and replace the join query with a simple unqualified select.

Suppose the foreign key constraint is removed. Does that change anything? No – a unique constraint on the address would be sufficient for us to conclude that the join means no duplication of rows in customer will take place. Therefore it can once again be replaced with the simple `select * from customer;`.

If, however, both constraints are removed and we cannot be sure that there is at most one address corresponding to a customer, then we have no choice but to do the join.

For an even wilder example, when an outer join occurs with distinct keyword: in a many-to-many relationship (the example in the source material is about actors and films) then outer join would produce duplicate tuples. The query asks for a listing of actor names. It is an outer join query (which I confess makes very little sense: why do you care about the films entered who have no actors in them, and why would there be any anyway?). Because it's an outer join you will even return the actors who appear in no films (also known as "waitstaff in LA restaurants"[17]). And thus potentially a lot of duplicates that are eliminated.

This last example is a bit forced, at least in my opinion. Why would you query all actors whether or not they had been in a film by referencing films? If you don't care whether they had been in a film, why do you even look at the films table... Anyway, this sort of thing could happen in an application where the SQL statement is composed by some if-statement logic where there are multiple checkboxes like "appears in a film" and "does not appear in film" and two conditions are added (like "incoming = true OR incoming = false").

Obviously, the more complex the query, the harder it is to determine whether or not a particular join may be eliminated. More than that, the same queries written exactly the same on a database in which the constraints have not been added would not be eligible for the join elimination optimization. In the inner join example, the foreign key and not null constraints, for example, are beneficial. This reveals a second purpose why constraints are valuable in the database. In addition to allowing us to enforce logical rules of the application inside of the database, it allows queries to be completed more efficiently.

Perhaps an analogy helps. You are asked to search through the library to find all copies of the book "Harry Potter and the pthread House Elves". That is a plausible task. But, suppose that you know as well there is a rule that this library will keep only one copy of that book ever. If that is the case, as soon as you have found the single copy of that book, you can stop looking (no need to check more "just in case"). This sort of optimization is very similar in that the rules let us avoid doing unnecessary work and that is a big part of the optimization routine.

The following short table tells us a bit about what database servers implement join elimination (hint: Not MySQL). The original is from [Ede17a] but I've modified it a bit to make it more colourblindness-friendly (mostly by making it grey instead of red and green):

Database	INNER JOIN: to-one	OUTER JOIN: to-one	OUTER JOIN DISTINCT: to-many
DB2 LUW 10.5	Yep	Yep	Yep
MySQL 8.0.2	Nope	Nope	Nope
Oracle 12.2.0.1	Yep	Yep	Nope
PostgreSQL 9.6	Nope	Yep	Nope
SQL Server 2014	Yep	Yep	Yep

For those databases that do not support automatic join elimination developers simply have to "do better".

[17] I apologize if that was overly mean.

Evaluation Plan Selection

It was perhaps oversimplifying to have said earlier that choosing a plan was just as simple as picking the one with the lowest cost. There is a little bit more to it than that. There is always a catch. There about choosing the one with the lowest cost is correct (generally) but the difficulty is in devising and calculating all possible evaluation plans. These operations are not free in terms of CPU usage or time and it is possible to waste more time on analysis than choosing a better algorithm would save.

A simplified approach, then, focuses just on what order in which join operations are done and then how those joins are carried out. The theory is that the join operations are likely to be the slowest and take the longest, so any optimization here is going to have the most potential benefit.

We already know that the order of joins in a statement like $r_1 \bowtie r_2 \bowtie r_3$ is something the optimizer can choose. In this case there are 3 relations and there are 12 different join orderings. In fact, for n relations there are $\frac{(2(n-1))!}{(n-1)!}$ possible orderings [SKS11]. Some of them, are obviously symmetric which reduces the number that we have to calculate, since $r_1 \bowtie r_2$ is not different from $r_2 \bowtie r_1$ (in relational algebra). In any case, even if we can cut down the symmetrical cases the problem grows out of hand very quickly when n gets larger.

Once more than three relations are affected by a join query it may be an opportunity to stop and think very hard about what is going on here, because this is quite unusual if the database design is good. The database server may want to ask why do you have a join query that goes across six or eight or twelve relations, but the database server (sadly) does not get to write the developers a nasty resignation letter saying that it can't continue to work this hard due to the negative effects on its health. It will dutifully do the work you asked it to and even try to make the best of this inefficient situation by optimizing it. But clearly it cannot examine all (non-symmetric) approaches and choose the optimal one. It would take too long.

Fortunately, we can create an algorithm that can "remember" subsets of the choices. If we have, for example, $r_1 \bowtie r_2 \bowtie r_3 \bowtie r_4 \bowtie r_5$ and the database server does not segmentation fault in disgust, we can break that down a bit. We could compute the best order for a subpart, say $(r_1 \bowtie r_2 \bowtie r_3)$ and then re-use that repeatedly for any further joins with r_4 and r_5 [SKS11]. This "saved" result can be re-used repeatedly turning our problem from five relations into two three-relation problems.

This is a really big improvement, actually, considering how quickly the factorial term scales up. The trade-off for this approach is that the resultant approach may not be globally optimal (but instead just locally optimal). If $r_1 \bowtie r_4$ produces very few tuples, it may be maximally efficient to do that join computation first, a strategy that will never be tried in an algorithm where r_1, r_2, and r_3 are combined to a subexpression for evaluation.

Remember though, this is as estimating process. The previous statement that said $r_1 \bowtie r_4$ produces very few tuples as if it is a fact. The optimizer does not know that for sure and must rely on estimates where available. So even though the optimizer may, if it had tried all possibilities, determined that $r_1 \bowtie r_4$ produces the fewest tuples and should be joined first, it is possible that estimate was off and the actual cost of a different plan was lower.

A simple pseudocode algorithm for using dynamic programming to optimize join orders is below from [SKS11]. In this, imagine that there exists a structure `result` that contains both a plan and a cost element. This result is stored in some array or other data structure for future retrieval. This recursive algorithm has $O(3^n)$ behaviour which is... well... it's not going to win algorithm of the year.

```
procedure find_plan( subquery S )
  if current subquery S result has already been computed
    return previously computed result for S
  end if

  declare variable result

  if current subquery S contains no joins
    set result.plan for S to best way of accessing this relation
    set result.cost for S this relation based on plan
  else
```

```
    for each non empty subset S1 of current relation S that is not equal to S
      variable r1 = find_plan( S1 )
      variable r2 = find_plan( S - S1 )
      A = best algorithm for joining r1 and r2
      cost = r1.cost + r2.cost + cost of A
      if cost less than current best plan for S
        result.plan = execute r1, execute r2, join using A
        result.cost = cost
      end if
    end for
   end if
return result
```

This is perhaps a bit hard to read and may require some explanation. The procedure checks if there has already been a result computed for this and if it is then there is no sense in re-doing that work. Use the already known result instead. Otherwise we will break the query down into smaller and smaller parts until we get down to just one relation: retrieving the matching part of that relation is computed. Then we can go up a level and find the best way to join that with the result (which itself may need to be broken down). This recursive procedure will eventually produce a result for how long it will take to perform the series of joins.

The sort order in which tuples are generated is important if the result will be used in another join. A sort order is called *interesting* if it is useful in a later operation. If r_1 and r_2 are being computed for a join with r_3 it is advantageous if the combined result $r_1 \bowtie r_2$ is sorted on attributes that match to r_3 to make that join more efficient; if it is sorted by some attribute not in r_3 that means an additional sort will be necessary [SKS11].

With this in mind it means that the best plan for computing a particular subset of the join query is not necessarily the best plan overall, because that extra sort may cost more than was saved by doing the join itself faster. This increases the complexity, obviously, of deciding what is optimal. Fortunately there are, usually anyway, not too many interesting sort orders [SKS11].

Generating Alternatives

Join order optimization is a big piece of the puzzle but it's not the only thing we can do in query evaluation. Let's briefly revisit the subject of how equivalent queries are formed. We already decided it is too expensive to try out all equivalent queries, but perhaps we are determined to try to at least generate lots of alternatives. The textbook [SKS11] summarizes some strategies from the Volcano research project that would make it possible to generate a larger number of plans than we might otherwise be able to. We need four things.

1. A way of storing expressions that reduces duplication and therefore keeps down the amount of space needed to store the various queries.
2. A way of detecting duplicate derivations of the same expression.
3. A way of storing optimal subexpression evaluation plans so that we don't have to recompute them.
4. An algorithm that will terminate the evaluation of a particular plan early if it is already worse than the cheapest plan found so far.

The details are obviously a bit complex. Most of this is beyond what we will discuss in this course, but some of those things we have already seen. For example, the algorithm earlier has got a way of storing evaluations of sub expressions so that we can re-use them. It could be modified to terminate earlier if, for example, the individual costs of `r1` or `r2` were already larger than the known stored plan.

Nested Subqueries

On the subject of generating alternatives, if possible, nested subqueries will be transformed into an alternative representation: a join query. To summarize the rather long story, if evaluated the "slow" way the subquery needs

to be run a lot of times. Thus, to make it faster, we would prefer to turn it into a join (which we already know how to handle). If really necessary we can run the subquery once and use that temporary relation in a join (where exists or "in" predicates may fall into this category).

Suffice it to say that transformation of nested queries is complicated and it may mean that the optimizer is unable to find some efficient route for executing the desired query if the query is complex. If possible, nested queries should be avoided. Sometimes you don't have much choice however. Amusingly, many years ago, MySQL did not support subqueries and that was one way to force people to come to grips with how to use joins. But somehow I do not think MySQL was intentionally doing anyone a favour, any more than having a terrible optimizer "forces" developers to write better queries.

Shortcuts

Now we will talk about some heuristic rules (guidelines, really) that we have definitely mentioned earlier when we talked about, for example, how to perform a selection. Now we can actually discuss them more formally.

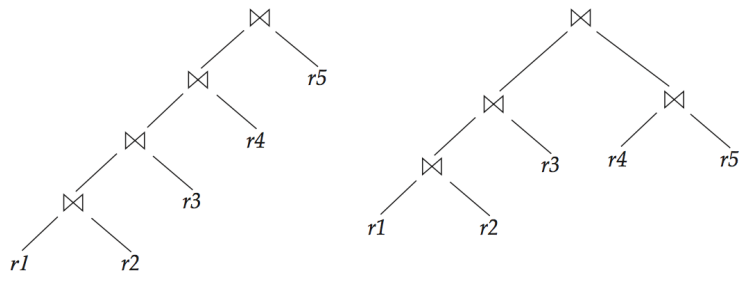
Perform selection early. No surprises here: the sooner we do a selection, the fewer tuples are going to result and the fewer tuples are input to any subsequent operations. Performing the selection is almost always an improvement. Chances are we get a lot of benefit out of selection: it can cut a relation down from a very large number of tuples to relatively few (or even one).

There are exceptions, however. One from [SKS11]: suppose the query is $\sigma_\theta(r \bowtie s)$ where θ refers only to attributes in s. If we do the selection first and (1) r is small compared to s and (2) there is an index on the join attributes of s but not on those used by θ then the selection is not so nice. It would throw away some useful information and force a scan on s; it may be better to do the join using the index and then remove tuples with the selection.

Perform projection early. Analogous to the idea of doing selection early, performing projection early is good because it tosses away information we do not need and means less input to the next operations. Just like selection, however, it is possible the projection throws away an attribute that will be useful. If the query does not ask for the join attribute in the output (e.g., does it matter what a person's address ID is?) then that join attribute will need to be removed from the output but if removed too soon it makes it impossible to do the join.

Left-deep join orders. Some query optimizers do not bother doing all the fanciful join optimization routines to solve which joins are best; instead they will consider join orders where each of the right operands of the join is always one of the initial relations r_k from the provided query $r_1 \bowtie r_2 \bowtie ... \bowtie r_n$. The reasoning behind this is it takes "only" $O(n!)$ time to consider all left-deep orders rather than all possible join orders [SKS11].

This is perhaps best illustrated with the following graphic, also from [SKS11]:



(a) Left-deep join tree (b) Non-left-deep join tree

In part (a) of the diagram the joins are all shown as happening in some ascending order from r_1 through r_5 but in reality the optimizer would consider this one possibility and then rearrange the relations through all possible arrangements to work out what is best.

Set limits. Another strategy for making sure we choose something appropriate within a reasonable amount of time is to set a time limit. Optimization has a certain cost and once this cost is exceeded, the process of trying to

find something better stops. But how much time to we decide to allocate? A suggested strategy from [SKS11] says to use some heuristic rules to very quickly guess at how long it will be. If it will be very quick then don't bother doing any further searching, just do it. If it will be moderate in cost, then a moderate optimization budget should be allocated. If it is expensive then a larger optimization budget is warranted.

Plan caching. In any busy system, common queries may be repeated over and over again with slightly different parameters. For example, [SKS11] suggests the following sort of query: a student wishes to query what courses they are enrolled in. If one student does this query with a particular value for student ID number, we can re-use that same evaluation plan in the future when another student does the exact same query with her student ID number instead.

The results will be different and this query may be more expensive on the second run if, for example, the second student is taking 7 courses this term and the first student is taking 5. That is expected, all we really needed was an estimate. With that in mind, if we have actually carried out the exact query we can use the actual execution time as the updated estimate for how long it takes (or perhaps an average of the last n executions of that same query if it has high variability).

21 — Practical Optimization with SQL

To wrap up the topic of query optimization we'll have a video. This goes over the advantages of use of SQL over Java (sorry Java programmers) and digs into the optimization process in a way that hopefully convinces you that it is worth your while to master SQL (and understand your optimizer) rather than try to fight through a lot of blocks of code to do it in your application.

The talk is entitled "How Modern SQL Databases Come up with Algorithms that You Would Have Never Dreamed Of" by Lukas Eder: `https://www.youtube.com/watch?v=wTPGW1PNy_Y`

Slides available at [Ede17b].

22 — Transactions

Transactions

Transactions are fundamental to database operations. A transaction is a grouping of operations that belong together and should be treated as an indivisible unit. You will recall from a great deal of discussion on the subject of concurrency that bad things can happen when an intermediate state of a multiple-step operation becomes inadvertently visible. Most of the examples we covered in earlier courses looked at things like `x++;` being a read, addition, and a write and how concurrent accesses to `x` could result in the wrong value being read or written. The solution we used then was mutual exclusion: forcing other accesses to `x` to wait while some operation was in progress. The transaction is the database way of dealing with the need to prevent intermediate state from becoming visible (although mutual exclusion can be used for a few purposes as we'll end up seeing later on).

By this stage you have almost certainly done something involving transactions for code management. Version control systems such as git and subversion use transactions. If you are making a large commit in subversion and the wireless network drops out, the commit is cancelled and from the view of anyone else using that repository, your half of a commit never happened. That's a big improvement over the cvs (concurrent versioning system approach) where the half finished commit appears in the repository, breaking the build. If your commit is successfully completed then the repository is updated and all is well. This seemingly-simple use case really captures all the most important attributes of transactions, as we will see. Understanding transactions as a concept is easy... implementing their behaviour is harder...

You may also recall from an operating systems class that modern file systems like NTFS or HFS+ are journaled. This is also a form of transaction management. A write to disk is treated as a transaction in that the updates to the file system metadata takes place inside a transaction meaning the metadata can never be left in an inconsistent state. For various reasons the actual writes of user data may not get that same treatment.

To execute a transaction we need first to create one. A transaction has some sort of begin transaction statement at the beginning, then the operations to take place in the transaction, and finally an end transaction statement. Execution looks something like writing down the transaction into a log, doing the operations in the transaction, and when the last one is complete, if all went well, marking the transaction as successful.

A transaction has a state and the actual implementation may vary from system to system but we will consider the following:

- **Not Started**: Transaction has been created but it has not started running yet (when we talk about concurrency control we'll see why one might be in this state for a while...)
- **Running**: Transaction has started running.
- **Ready to Commit**: Also sometimes called partially committed; all operations are finished but it is not yet committed and permanently recorded.
- **Committed**: Transaction has finished and the effects have been recorded in the database permanently.
- **Failed**: A problem has been encountered in the execution of the transaction and it needs to be rolled back.
- **Aborted**: Transaction has finished unsuccessfully and any changes made so far have been undone.

How might a transaction be unsuccessful? We have already examined one situation in the context of subversion, that is an operation in progress gets interrupted by a network disconnection on the client side. But there are many more possibilities; the server could crash, or, more interestingly an error might be encountered in execution of the query. A division by zero or foreign key constraint violation will move the transaction to the aborted state immediately and halt all further execution of that transaction, rather like encountering an exception in a language like Java. These exceptions, however, tend to be handled by undoing any partly-done changes.

Transactions should have the *ACID* properties [EN11]:

- **Atomicity**: A transaction is atomic (indivisible) – it either is completed in its entirety or it is as if it never happened at all.
- **Consistency preservation**: If the transaction is completed successfully, it moves the database from one consistent state to another consistent state (i.e., does not break any key constraints etc!)
- **Isolation**: A transaction should appear as if it is the only transaction executing right now even though it's possible many transactions are executing concurrently.
- **Durability**: A transaction's changes should be persistent in the database and not be lost.

The good news is that we have already seen a number of the important concepts for achieving the ACID properties in previous courses. We have already learned a bit about how to ensure atomicity of larger operations as well as a little about durability.

Obligatory, Mandatory, Non-Optional Bank Analogy

It was inevitable that a bank analogy would be applied to transactions. There's just kind of no way around it. All the textbooks do it. And why not, it works well enough for our purposes. The example from [SKS11] revolves around transferring $50 from account A to account B. This is a common enough operation: your friend buys tickets to that Metallica Concert[18] you want to go to, and you send an electronic bank transfer to said friend in the amount of your half. Here's a situation where you need both parts to happen for everyone to be happy. Your friend will not be happy if the money is deducted from your account but never arrives. The bank will not be happy if the money appears in your friend's account but was never deducted from yours. We need a transaction.

If we wanted to write out the transaction in some sort of pseudocode, ignoring for the moment that we need to turn this into database statements, the steps are:

```
BEGIN TRANSACTION
read A.balance
A.balance = A.balance - 50
write A.balance
read B.balance
B.balance = B.balance + 50
write B.balance
END TRANSACTION
```

Those who are familiar with banks will note that the bank, of course, takes the money out of your account first. Those of you familiar with my lectures will note that it sure looks like I hate banks.

Let's think about the ACID principles as they apply here.

Atomicity. If we do not have atomicity in this transaction then it is possible that some halfway completed state is revealed, such as after the deduction has occurred from account A but before the increase in account B. That would be inconsistent, and we do not want that state to be visible (see: isolation). Moreover, if we have a crash at this stage, the operation is only partway complete and that is undesirable. But we already wrote A.balance (and let's just say for now that write goes out to disk).

[18] Somehow I doubt their tickets are as cheap as $50 but let's pretend.

The way we prevent this problem is the same way that NTFS does it (if you remember that part of the operating systems course): there is a log file. The transaction steps are written in the log file before they are executed. The transaction is treated like a checklist: after each operation is done we check off the box next to the operation and then we may proceed to the next. If there is a crash and we have an inconsistent state, we can use the log to go backwards and undo the partially complete operation. And, as something new for database transactions, we might even be able to pick up where we left off and finish it. Either way, we can't leave it half done: forward or back, we have to be at one consistent state or another.

Consistency. There aren't any sort of foreign key or unique key rules here but there does exist a consistency rule: the sum of account A and account B must remain the same after the transaction is done. If that is not true, then something is wrong and money has gone missing or been created out of thin air. Destroying and creating money are the job of the central banks and they get very upset when you try to do it yourself at home.

There could exist other consistency rules in this system that come in to play. For example, the balance may be required to be non-negative. So if you attempted to transfer $50 to someone from an account that contains only $31.25 then the transaction will not succeed because the consistency check that says balance must be greater than or equal to zero will be violated. Banks do not necessarily follow this rule in real life because they have the ability to overdraft (i.e., temporarily allow your account to go into the negative) but overdraft has a limit as well, so if you are allowed overdraft of $50 then your account may have a balance of -$50.00 but not lower and the same sort of rejection of the update would occur if that constraint is violated.

Isolation. As we know, when the same data is being accessed by multiple threads in a C program, there is the possibility that an error will result. An unlucky interleaving of statements could happen here as well. In a simple model, we can just solve the problem by doing transactions one at a time. This has adverse impacts on performance but guarantees the isolation property. Some other strategies that we have previously learned, such as mutual exclusion, locking will come into play here.

Isolation is a very complex subject that will be the subject of the next lecture. On top of that, we will need to also dig into the topic of concurrency control in the database which is, again, an upcoming topic.

Durability. Finally, once the transaction has reached the end statement and it is ready to commit, that change should be permanently reflected in the data on disk. For a transaction to be truly durable it means that if the system crashes at this point, the transaction's changes will not be lost. In short, either (1) the data has already been written to disk or (2) the information already written to disk is sufficient that we could redo the changes if the system is restarted after a failure [SKS11].

Digression: Stable Storage

For a transaction to be really considered permanent, it might not actually be enough just to write it out to disk. Disk is not volatile storage and data on disk will be preserved if the power goes out or the server process crashes or something, but that does not account for another mode of failure: hard drive death! Hard drives can and do die at the worst possible times. Far, far too many people go around these days acting as if their drives will never die and the single copy of important things (pictures, documents, whatever) will always be accessible. If you do not believe that hard drives die, I will be glad to show you one that died on me in April of 2015. With no warning.

Ultimately I wasn't very upset about the loss of that drive even though it contained installed programs, my music collection, et cetera. Why not? I had backups of everything that could not be obtained again easily (e.g., just download the game again from Steam).

In [SKS11] a further level of storage beyond non-volatile is called *stable storage* which they define as where data can never get lost. Never is somewhat exaggerated: no matter how many forms of storage you have, a nuclear war will probably get them all (or at least all of us), and if nothing else, the heat death of the universe will certainly finish them off. Physics can actually be kind of depressing. Maybe we are digressing. The point of stable storage, though, is to be sure that even the failure of nonvolatile storage is not sufficient for data to get lost, and a transaction isn't really truly durable until it has been written into the stable storage.

In real life where we don't have infinite money and plan for more reasonable scenarios than nuclear armageddon

or the rise of Skynet[19] there are a few common techniques for creating (an approximation of) stable storage.

The first is the RAID array (redundant array of independent disks) so the failure of any individual disk does not result in data loss. RAID has several levels and some of them offer no redundancy at all, others offer a lot more. Choose wisely. I will also note, even though it is a bit of a digression, that it's a super bad idea to buy all your RAID disks at the same time from the same manufacturer. They likely came from the same lot and will tend to live about equally long. So if you lose a drive, you think this is recoverable, only to discover that the strain of rebuilding the data from the other drives in the RAID array has killed another of the drives and now your data is lost. Not good.

The next strategy is backups: having another copy of the data on an independent storage medium. That is not to say that the original and the backup cannot be both hard drives, just that they need to be two separate hard drives. If you don't have backups and a drive has died you can send it out to a data recovery service and they are not cheap and cannot guarantee success. Really. Buying a backup drive today is like a tenth of the price of sending your dead drive out for recovery.

For best results, the data you consider important should be backed up and there should also be off-site backups. There are stories of companies where they were good about taking backups but the backups lived in the same server room as the original servers and the original and backup copies were both lost in a server room fire.

For the strictest possible definition, a transaction is really only durable when it is not only written to stable storage, but the backup copies of the database have been updated as well to reflect the change. It doesn't get any more permanent than that.

Atomicity and Durability

We can also take some time to take a closer look at atomicity. We know the key idea and how it should work, but have only gone over briefly how it actually takes place. We will examine two approaches.

The log file is the most important key to the usual approach. The more detail is written into the log file, the more options we have for dealing with a failure of the transaction if it occurs. The trade-off here is of course the size of the log and the amount of time spent maintaining that log. A sample log entry from [SKS11], represented here as a table structure:

Transaction ID	Relation	ID	Attribute	Old Value	New Value
385	account	8675309	balance	5000.00	4950.00

The table headers probably don't appear in the log and it may be binary data rather than especially human-readable. Nevertheless, the idea is hopefully clear. By showing the old and new data we can easily check if the data was updated before the crash and whether we need to set it back to the old value if the transaction did not succeed.

Another way that we can get atomicity is the ZFS (A file system developed by Sun Microsystems) approach. In this case when a transaction is to be executed, a copy of the data is made. The copy is modified, and if the transaction runs to completion, then the copy replaces the original data.

That is a neat approach but comes with some drawbacks, such as the fact that in ZFS if the file system gets full you can never delete anything to make more space; or in other words you do need a significant amount of extra space, and accordingly, a lot more reads and writes to accomplish the same thing. The upside is that if the transaction is to be aborted, we just throw away the copy and that is very efficient; nothing needs to be undone. Optimizing for the failure case, however, maybe seems overly pessimistic. We would expect that most database operations succeed rather than fail the log approach seems a bit better if we think that will be the case.

Getting to the state "committed" actually means that the transaction is not only completed but written out to disk in such a way that if the system crashes now that transaction will not get lost. That is an important rule to hold to because without it we could check off an item as done when it is not actually done.

[19] If a very large-muscled guy with an Austrian accent ever says to you "Come with me if you want to live.", you go.

If the transaction succeeds and is committed then it can be removed from the log. This keeps the log file size reasonable. In many cases the log contains only the in-progress transactions (i.e., the committed transactions are removed right away) so a quick glance at the log file will tell the database server on boot up whether any transactions were in progress at the time of crash.

Once a transaction is committed, we cannot cancel or roll back its effects. In the words of Lady MacBeth: "What's done is done and cannot be undone." You can return the system to an equivalent state to the before by executing a *compensating transaction* [SKS11]. If the bank makes a mistake and withdraws $50 from your account, and this needs to be corrected, it can't be corrected by annulling the initial transfer. Instead, the compensating transaction would be the opposite: deposit $50 into your account to put you back into the state you were in before. Sometimes, however, there is no going back! Compensating transactions are not always possible when information is tossed away such as a deletion or truncate statement and are generally left to the user to fix. You took backups, right? You can restore data out of those...

Suppose that a transaction does not succeed. There are now two ways forward from the aborted state. The first is to re-try the transaction. Sometimes the transaction failed due to temporary circumstances, network disruption being one such example. As we will see, transactions sometimes interfere with each other a bit and that can cause one of them to need to start over again. The transaction, having been aborted, is assigned a new number and it tries again. It is worth noting that there is likely a need for a limit on how many times a transaction may be re-started this way, otherwise an unsuccessful transaction may be doomed to constantly retry and fail for all eternity, wasting database resources.

The alternative is to kill the transaction, which is really just the same as giving up. Giving up is mandatory if the operation in question failed due to some unrecoverable error such as trying to violate a constraint or any other reason the transaction can never succeed. An error of some sort will be returned the requesting program/client to indicate the failure. Once that happens it is out of the database server's hands and purview; the user may try again or the application program may have its own error handling routines to deal with this.

Writing to console and network. Another concern raised in [SKS11] has to do with external writes, i.e. making results available to users and other devices or individuals outside the database system. Once data has gone out over the network or has been printed to the console there is no way to get it back. The fully correct rule to handle this would be to allow no such output until the transaction is committed; we could write data to a temporary location until such time that is true.

Sometimes the state of an external write of some sort is fairly indeterminate: what happens if we aren't sure whether the data was sent to the printer? Do we try printing again? Probably not, the user will either print again manually if necessary or just give up and use some other printer if it doesn't work.

Sometimes, sadly, we actually want to show the status of a transaction to a user. Users love progress bars. For good reason. An operation that will take a long time may seem concerning to a user, who fears that it is stuck. For this reason and some others we might need to allow long running transactions to be interactive. That comes with its own headaches...

23 — Transaction Isolation

Transaction Isolation

Recall from last time the idea of transaction isolation is that multiple concurrent transactions should not be able to lead somehow to an inconsistent state due to concurrent operations. The simple approach is to force transactions to run one at a time, which of course ensures that concurrency is not a problem, but also ensures that nobody uses your database because it's very slow. Previous courses have examined in some detail why you want concurrency: better performance, better resource utilization, fewer user tears. Let us waste no time discussing the particular merits of concurrency.

The database server has the same threading problems that you likely recall from the discussion of concurrency: the operating system schedules the threads and it chooses when it is a good time for a switch between them, meaning the database does not have control over when such a switch will happen. So if two operations are in flight at a particular time, they could be executing in two different threads and switches between them can occur at arbitrary times. If anything, they will occur at the worst possible times and really cause problems.

Let's imagine that we have two transactions that are going to execute, in an example from [SKS11] I've modified a bit to be a little less strange. In the original there is a transfer in the amount of 10% of the balance from one account to another but this does not seem sensible since one normally specifies an amount to be transferred, not a percentage. In the modified example, one transaction is a transfer of $50 from A to B, and another transaction is the calculation of monthly earned interest, wherein the lucky account holder of A receives a humongous 1% interest on the balance in this account.

Transaction T_1

```
T1.1. read A.balance
T1.2. A.balance = A.balance - 50
T1.3. write A.balance
T1.4. read B.balance
T1.5. B.balance = B.balance + 50
T1.6. write B.balance
```

Transaction T_2

```
T2.1 read A.balance
T2.2 A.balance = A.balance * 1.01
T2.3 write A.balance
```

This sort of multi-column pseudo code concurrency problem may be reminding you of unpleasant memories of learning about concurrency. Well, we still have the usual problem here. An improper interleaving of two statements means one of the changes gets lost, and the $50 is not deducted or the interest is not calculated at all.

This pair of transactions is interesting in that there are two correct answers. Suppose the initial value in account A is $2000. If the transactions are executed serially, first T_1 then T_2, then the $50 is deducted and the interest is calculated on $1950 and the total amount is $1969.50. If the transactions are executed serially but in the opposite order, then the interest is calculated on $2000 and then the $50 is deducted, meaning the total balance at the end is $1970.00, a difference of 50 cents. Both answers are correct in the sense that they are consistent.

These two execution sequences are called *schedules* and they represent a chronological ordering in which instructions are executed [SKS11]. A schedule is called valid if it does not break any of the rules and cannot cause some inconsistent state. A serial schedule means one transaction runs to completion before the next begins and for n transactions there are $n!$ different valid serial schedules. Probably you have noticed however, that the operations

on the A account are finished in T_1 after statement `T1.3` and you could in theory start the second transaction a little bit early to get some performance increase. Well – maybe! That might be revealing the partial state of a transaction and that might not be acceptable; what if an error is encountered on assigning the balance of B because and the transaction is to be rolled back? Things start to get interesting...

We do not actually need for all transactions to be done serially, but they do need to be *serializable*, which means there must exist an equivalent serial schedule [SKS11]. That means the end result when all is said and done must be the same as if the transactions executed in a serial order (any serial order). So the following order is acceptable because it is equivalent to the serial schedule of first T_1 and then T_2:

```
T1.1. read A.balance
T1.2. A.balance = A.balance - 50
T1.3. write A.balance
T2.1 read A.balance
T2.2 A.balance = A.balance * 1.01
T2.3 write A.balance
T1.4. read B.balance
T1.5. B.balance = B.balance + 50
T1.6. write B.balance
```

But the following order is not okay because it does not match either T_1 then T_2 or T_2 then T_1:

```
T1.1. read A.balance
T1.2. A.balance = A.balance - 50
T2.1 read A.balance
T2.2 A.balance = A.balance * 1.01
T2.3 write A.balance
T1.3. write A.balance
T1.4. read B.balance
T1.5. B.balance = B.balance + 50
T1.6. write B.balance
```

If this order were executed, the balance would be $1950.00, meaning the calculation of the interest would be lost altogether, an unacceptable outcome. And looking at the value of $1950.00 we can verify it is neither $1969.50 nor $1970.00 and is therefore equivalent to neither schedule.

Determining Serializability

If our intention is to ensure serializability, we do need to know how to tell if a schedule is serializable. For simple examples like above, we can determine by inspection whether it is serializable and if pressed to do so on an exam could back that up with the reasoning showing that a certain schedule produces the same results as T_1 then T_2. Those are the simple cases; when multiple transactions are interleaved it starts to get more difficult.

From the view of the database (or any program, really), it is more or less irrelevant what the contents of a variable or attribute are. The order reads and writes are what matter; the actual value is pretty much irrelevant. In the previous examples, it changes nothing if the amount transferred is $5 or $500, or if the interest rate is 1% or 3%, or if the starting balance is $1000 or $2000. It is the read and write operations themselves that need ordering.

In the textbook [SKS11] they are called something else but to use terminology consistent with the other courses that touch on this sort of conflict, we call them “hazards” and there are four possibilities.

1. **RAW** (Read After Write) - The classic form of dependency. The read has to take place after the write, otherwise there’s nothing to read, or an incorrect value will be read.
2. **WAR** (Write After Read) - A write cannot take place until the read has happened, to ensure the read takes the correct value.

3. **WAW** (Write After Write) - A write cannot take place because an earlier write needs to happen first. If we do them out of order, the final value may be out of date or otherwise incorrect.
4. **RAR** (Read After Read) - No such hazard!

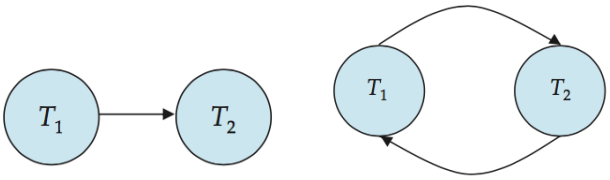
Yup, reads don't interfere with one another in any way, and why would they; that one transaction reads the value has no impact on a future read. But writes always cause a hazard. Two transactions T_i and T_j are said to *conflict* if they are operations by different transactions on the same data, and at least one of those operations is a write. It does have to be the same data, of course, otherwise there is no conflict.

Non-conflicting instructions can be swapped with no consequences in the schedule. If a schedule S can be transformed into S' by a series of swaps of non-conflicting instructions, we say that S and S' are conflict-equivalent [SKS11]. If a particular schedule S is conflict-equivalent to a serial schedule then it is called conflict-serializable [SKS11]. In general we need a particular schedule to be conflict-serializable, not serial; that is enough to make sure we can perform the operations and get consistent results.

Determining conflict-serializability. A simple approach to determining the conflict-serializability of a schedule relies on building a directed graph called a precedence graph. This is formed out of the transactions participating in the schedule and constraints are reflected as edges in this graph. Each transaction is a node. If transaction T_i does a read before transaction T_j does a write, that is an edge from T_i to T_j. The same holds for a read after write dependency and for a write after write dependency.

If this sounds a lot like a resource allocation graph, it's because it is. The resource allocation graph was used to detect deadlock by looking for a cycle. Here, we will still search for a cycle, because the presence of a cycle means that the schedule is not conflict-serializable. As before, known graph theory algorithms are perfectly suitable for detecting a cycle.

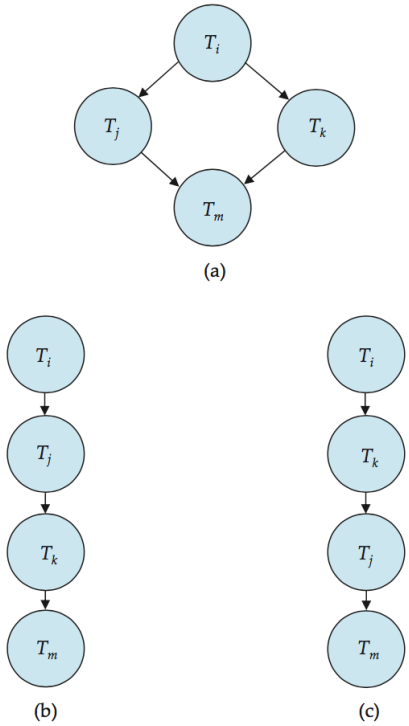
The following two graphs show very simple precedence graphs for two transactions [SKS11]:



Once the precedence graph has been assembled, it is fairly easy for humans to figure out a valid serial ordering of the transactions. Topological sorting is the name of the graph algorithm to get a linear ordering of the transactions. Remember that the sort order of the transactions does not mean that they are actually executed in that order completely; it is just that the execution is conflict-equivalent to the linear execution order.

If a cycle is detected, a topological sort is not possible: it cannot be turned into a linear ordering of transactions. That should be fairly obvious, but needs to be noted.

A diagram showing a precedence graph and two valid topological sorts is shown below [SKS11]. Suppose we have four transactions. From that, a topological sort is performed to get the linear order of transactions.



Both (b) and (c) in this diagram are valid serial schedules. They are not equivalent, but that is okay, they do not have to be. Either option is fine when it comes to executing the transaction. They may or may not result in the same final state, but they will always result in consistent state.

Isolation and Atomicity

If we have a schedule that is not serial but conflict-equivalent to one, we have a potential problem that we did not have before. Suppose we have two transactions that are running in parallel and we have a schedule that is conflict equivalent to first T_1 then T_2. What if transaction T_1 writes a value, then T_2 reads that value, but T_1 aborts immediately afterwards and needs to be rolled back? For example, T_1 subtracts the \$50 and T_2 calculates the 1% interest; if the subtraction transaction is aborted, then the calculation of interest may not use the updated value in its computation. But what if it has already read it? We have a problem!

We would like to know what schedules make recovery possible at all, and even which schedules make recovery relatively simple [EN11]. If we can recover from a problem, the schedule is called *recoverable*. A schedule that is not recoverable should never be chosen by the database server because it means a transaction that is aborted could potentially result in inconsistent results, violating the atomicity principle.

On the other hand, if a transaction T_k is committed, it should never be necessary to roll that transaction back [EN11]. If we did, the durability property would be violated. Commit is final and we do not want that policy to be violated.

The behaviour we are looking for in the example is that if T_1 aborts for some reason, T_2 must also abort. The second transaction is dependent on the first. Thus T_2 cannot commit until T_1 has committed. The general statement of this principle is that the schedule is recoverable as long as no transaction T commits until all transactions T' that precede it have committed. "Precede" in the previous sentence refers to our idea of the precedence graph.

The placement of the commit operations is what makes a schedule recoverable. So in addition to paying attention to the placements of the read and write operations, the placement of the commit operations matters too. All of this is distinct from the idea of actually performing the recovery; how the transactions are rolled back is independent of the fact that we need to actually perform that rollback. If sufficient information is kept in the log then it does make the rollback possible (but not very pleasant).

If we have two transactions as above like T_1 and T_2 and the first transaction aborts, it may cause the second transaction to abort. That may not sound like a problem, but if there are a lot of transactions in flight at the

same time, a transaction that aborts may cause some other transactions to abort, which would in turn cause more transactions to abort. This is called a cascade rollback. This can easily cause a lot of time to be wasted since so much work needs to be done to undo the transactions that cannot be completed.

It may be desirable, where possible, to prevent the need to cascade rollbacks. A schedule will be *cascadeless* (i.e. contain no cascade rollbacks) if every transaction only reads items that have already been committed. It should be easy to convince yourself that if that is the case, then a cascade rollback can never occur. If transaction T reads an item written by T', then either (1) T' commits, in which case the transaction cannot abort and there is no problem, or (2) T' aborts and the read by T has not yet taken place. It does delay T but ensures no cascading rollback. There is, of course, potentially reduced performance of normal transactions to avoid a very costly cascade rollback.

There is a third, even more restrictive schedule called a *strict schedule* in which transactions may neither read nor write a data element until the last transaction that writes that element has committed (or aborted) [EN11]. This even more restrictive schedule means that it will be relatively easy to recover if we must. Suppose the field x is initially 1 and T_3 writes 3 into x and then T_4 writes 5 into x. If T_3 aborts the rollback might set the value of x back to 1 meaning the T_4 change is lost. A strict schedule avoids this.

As a strict schedule is defined as an additional restriction on a cascadeless schedule, it is trivial to show that the strict schedule is also cascadeless. And a cascadeless schedule is always recoverable (also trivial to show).

Transaction Isolation Levels

Serializability is how the database server ensures that data operations produce consistent results. As there are performance costs that come with performing things strictly, we will sometimes allow a little bit of rule-breaking to allow better performance... This is dangerous, and it means that more care needs to be taken on the part of the developers: if they want to use this sort of feature they need to be responsible about its use.

The isolation levels specified in the SQL standard are [SKS11]:

- **Serializable**: ensures serializable execution (although some databases don't necessary fulfill this...)
- **Repeatable read**: allows only committed data to be read and further requires that if the transaction reads a data item x twice, those two reads may not have another transaction that writes x between them.
- **Read committed**: allows only committed data to be read, but does not require repeatable reads.
- **Read uncommitted**: allows uncommitted data to be read.

All isolation levels above also forbid writing a data item that has already been written by another transaction that has not yet been committed (or aborted). The transaction isolation level is often set at "read committed" but it can be changed by some syntax in SQL if the application developer wishes to allow better performance [SKS11]. We will come back to the subject of how to actually implement the isolation levels in the next topic.

Non-repeatable reads tend to happen in real life a lot. An example from [SKS11] illustrates using the common example of selecting seats on a flight, but it could also apply to buying concert tickets, or online shopping. When you go to select seats, you are presented with a map and you get to make your choice about which seat. If multiple people are doing the same thing then there is no problem unless two people are trying to choose the same seat. If that's not the case, then there is no conflict. If they are choosing the same seat, there might be a slight problem: both people cannot have the same seat, no matter how much the airlines would really really like to sell the same seat twice.

Ultimately, one user will succeed and the second user will not. The successful user will be told of the success and the unsuccessful user will be prompted to select again, and be presented with the updated map. This is a form of rollback, really: the potential changes are not committed and anything attempted is undone; the user then may choose whether to give up or try again. Chances are the user does try again (they still want to get on that flight, don't they?) but they will need to make some changes before submitting again. In a rather unpleasant scenario, the user may be forced to try several times before succeeding.

We could force serializability by allowing only one passenger to select seats at a time. That might be acceptable (if annoying) if seat selection took place only at the time when flights are booked. That isn't what happens, though; people select them at check-in. In that case we let the users make edits, but the process of actually updating the database runs in transactions, and those transactions need to be serialized.

This problem is a general one when we have client applications interacting with data stored in the database. The stored data is called up and sent to the client for editing. If some other client is accessing that data at the same time, there is a conflict. Before changes are applied, it is common to reload the underlying data and see if it has changed; if so then the current edit cannot go through without some sort of confirmation or changes. Serializability occurs only on the parts that happen in the database; the user changes are all considered concurrent (for our definition of concurrent in previous classes: no defined order).

To actually achieve the isolation levels, there are three strategies in [SKS11] that we can discuss: locking, timestamps, and snapshot isolation.

Locking. Locking is pretty much what you would expect: synchronization primitives are established and then they need to be acquired and released as is appropriate. Locks may be implemented at various granularities: one lock for the whole database, one per table, or perhaps one per tuple (but that is a headache).

Suppose we have locks for each table. If we need more than one table, we have the possibility of a deadlock. Remember from the discussion of concurrency the idea of two-phase locking. A transaction attempts to acquire all the locks it needs and is informed of success or failure for each; if it fails to acquire any of them, it will release the ones it has acquired and then try again.

We can also have reader-writer locks which could speed up execution of the transaction as well, especially if reads are much more common than writes.

Timestamps. Timestamps are pretty simple; each transaction is assigned a timestamp, typically when it begins. Each data item has a read-timestamp and a write-timestamp, which are updated when the item is read our written respectively. The database server can then ensure that transactions that conflict are executed in order of the timestamps, and, failing that, aborting (and possibly restarting) the transaction.

Snapshot Isolation. We have already, to some extent, discussed the idea of snapshot isolation. When a transaction is to begin, a copy is made of the data needed and the copy is modified, and then finally the modified version is committed (or aborted). This avoids the need for locking of tables, since each transaction gets its own copy of the data.

This does have a potential drawback: too much isolation! If there are two concurrent transactions, it may happen that neither transaction can "see" the updates from the other, and the final state is then inconsistent in some way (i.e., not equivalent to a serializable execution). That is something we will see in the discussion of concurrency control; our next topic.

24 — Concurrency Control

Concurrency Control

Having discussed in some detail the idea of transaction isolation, we now need to turn our attention to the question of how the database will control the various concurrent elements to make sure that execution goes as planned. There are a variety of schemes, but at this point the ones we will examine will always assume that failure does not happen. Once we understand the concurrency control mechanisms we can then build on them to include failure recovery.

No Concurrency Control

It is possible from the point of view of the database to simply choose not to implement concurrency control of any sort. This is easy for the database designers and effectively means putting the problem of concurrency management in the hands of the users of the database (application developers).

This is not as insane as it sounds, since the application itself could have its own sort of concurrency control mechanisms and it would be fine to just let the application manage that. The solution is, however, inadequate for commercial databases since they can allow multiple applications to run at once and do not trust that application developers do things the "right" way at all times.

Lock-Based Protocols

As you will recall from learning about concurrency and synchronization, our typical "go-to" strategy for dealing with concurrency problems is locking of some sort. You are surely familiar with locks and locking so we will not need to repeat any of the material on that subject from earlier.

For performance reasons we will assume that we use reader/writer type locks, meaning that there are accordingly two modes in which a data element may be locked: shared mode, and exclusive mode. In keeping with [SKS11] the shared mode lock is denoted by S and exclusive by X. If transaction T_i has a shared lock on some data element e then it may read but not write it; if the transaction has an exclusive lock then it may both read and write. Exclusive is exactly what it sounds like in that no other transaction may have any sort of lock if one transaction has an exclusive lock.

Every transaction must request a lock before accessing a data item, and the sort of lock it will request depends on what it would like to do. The request is made of the concurrency control system and it will either grant the request or delay that transaction until the request is granted. This you will recognize as being the same sort of behaviour of the mutex and semaphore: a requestor may proceed or will be blocked for some period of time.

As for whether the request can be granted, this is pretty simple. If there are no locks for that item, then the request may be granted. If the request is exclusive and there exists any lock on that data item, the request must wait. If the request is for a read and there exists an exclusive lock on the item, the request must wait. If the request is for a read and there exist only read locks on the item, the read may proceed. If the request can be granted, that might not guarantee it will be granted immediately.

If there are more lock modes (which we will not consider in this course, at least) then we would need to check

to see whether a new request's locking mode is compatible with the existing ones. The compatibility rule for reader-writer locks is pretty simple as above, but in the more general approach it is necessary to check all the locks currently held for that item and see if they are compatible. If they are all compatible, the lock may be granted.

As with the kinds of locks we are accustomed to from C-like languages, there is also an unlock operation to, well, unlock that item. Unlocking an item will allow some other waiting transaction (or transactions) to be able to then lock that item.

The standard advice in concurrency is to unlock items as soon as possible (but no sooner). The sooner a transaction unlocks things it is finished with, the sooner another transaction may be unblocked and may proceed. But there is such a thing as unlocking too soon. Why? Serializability may not be ensured: it is possible that an early unlocking allows some intermediate state to be read. Let's do an example from [SKS11].

Let us suppose there are two concurrent transactions going on. T_1 is to transfer \$50 from chequing (account C) to savings (account S). The second is to show your total net worth which is computed by summing C and S (and we will ignore any other details like credit card debts, loans, investments, etc). Let's look at the statements needed to get this done:

Transaction T_1

```
T1.1 Exclusive lock C
T1.2 Read C
T1.3 C = C - 50
T1.4 Write C
T1.5 Unlock C
T1.6 Exclusive Lock S
T1.7 Read S
T1.8 S = S + 50
T1.9 Write S
T1.10 Unlock S
```

Transaction T_2

```
T2.1 Shared lock C
T2.2 Read C
T2.3 Unlock C
T2.4 Shared Lock S
T2.5 Read S
T2.6 Unlock S
T2.7 TEMP = C + S
T2.8 Print TEMP
```

If executed serially we have no problem; we will always get consistent results. No matter what order we execute the transactions, the result will be the same since any amount moved from C to S does not affect the total. If they are, respectively \$100 and \$200 then the total must remain \$300. This is fine. But if the transactions are executed incorrectly, we will print a result of \$250. How? Think back to earlier courses on concurrency: try to find an interleaving of statements that could lead to this inconsistent result.

A quick analysis says this can happen if T_2 reads the values of `C` and `S` after the decrease has been completed but before the increase. One such possible ordering places all of T_2 between statements `T1.5` and `T1.6`, but you can likely find other orders that reproduce this problem. As we know, any such order with a problem means we need to make some changes to prevent this problem.

How can we fix this? In short, the unlocking of elements needs to be delayed appropriately to ensure serializability: that is to say that partial state should not become visible. A modified version of T_1 and T_2 shown below that avoids the problem altogether.

Transaction T_1'

```
T1.1 Exclusive lock C
T1.2 Read C
T1.3 C = C - 50
T1.4 Write C
T1.5 Exclusive Lock S
T1.6 Read S
T1.7 S = S + 50
T1.8 Write S
T1.9 Unlock C
T1.10 Unlock S
```

Transaction T_2'

```
T2.1 Shared lock C
T2.2 Read C
T2.3 Shared Lock S
T2.4 Read S
T2.5 TEMP = C + S
T2.6 Print TEMP
T2.7 Unlock C
T2.8 Unlock S
```

There are no free lunches, though: the risk of longer-held locks is that there can be, as you guessed... *deadlock*! Deadlock, as you will recall, is what happens when two or more transactions (formerly processes) are permanently blocked waiting for one another. As before, an unfortunate interleaving of lock and unlock statements can mean that the transactions get stuck. Deadlock is bad, but not as bad (we think) as inconsistent states being shown (or worse, saved!). If a deadlock occurs, we can detect it and then can do something about it; if an inconsistent state is shown then we may never know about it.

There are often rules in the system for how transactions behave with respect to locks called the *locking protocol*. The locking protocol sets up some rules about when items may be locked and unlocked which reduces the number of possible schedules such that the only allowed schedules are conflict-serializable schedules [SKS11]. A schedule is considered legal if it follows all the rules of the locking protocol.

Alongside the risk of deadlock, there is also the possibility of starvation. As you will recall, a transaction (thread) starves when, even though it is ready to run, it never gets a turn because the other transactions (threads) are using the resource(s) that it needs. The more locks a particular transaction requires, the more likely it is to have to wait and the higher the risk of starvation.

One possible solution to preventing starvation as suggested by [SKS11] is to introduce a new rule about when a lock request is granted. The new rule is that the locks should be granted in order of request, meaning that a transaction T_i will never have to wait for a transaction T_j that requested that lock later. This is, obviously, in addition to the usual rule that the lock can only be granted when the requested lock mode is compatible with the existing locks, if any.

This new rule differs, in some way, from the kind of logic we have discussed in concurrency. In concurrency we have said that there is no guarantee as to what order waiting threads will be unblocked when the resource they are waiting for becomes available.

You will also recall the idea of two phase locking: we try to acquire locks and if we are unsuccessful, release any that we have acquired and try again from the beginning. The database two phase locking protocol is a little bit different. In this there are two distinct phases: growing and shrinking. In the growing phase a transaction may acquire locks; in the shrinking phase a transaction may release locks but may not obtain any new ones. The initial state is the growing phase and as soon as the first lock is released then it is not permitted to obtain any new ones [SKS11].

Any two phase locking protocol gives us, automatically, conflict serializability. For any transaction, there exists the point where the last lock as been obtained, and this is called the *lock point* [SKS11]. We can order transactions based on their lock points and it gives a serial ordering of the transactions. But that's all it gives us: not any sort of freedom from deadlocks or starvation.

There exist two more restrictive locking protocols in [SKS11]. The first is the strict two phase locking protocol, which requires that all exclusive locks are held until the transaction is committed. This guarantees that no intermediate states are ever made visible. The other is the rigorous two phase locking protocol which requires all locks, no matter the type, to be held all the way until the commit. With rigorous two phase locking, transactions are easily serialized by commit order.

We can try to squeeze out some more parallelism by allowing locking to have two more operations: a transition from shared to exclusive and from exclusive to shared. We still need exclusive lock to modify a value, but we can allow some values to be shared during the execution of the transaction. Still, moving from a shared to exclusive lock is called an upgrade, and that can only be done during the lock acquisition phase. Moving from exclusive down to shared is called a downgrade and it can only be done in the shrinking phase. An upgrade is like trying to acquire a lock, so it can result in blocking of a transaction.

As you might imagine, when presented with a list of instructions, a human can consider the transaction (or transactions) presented and make some sort of hopefully good decisions about how to put the lock and unlock statements in the transaction. Doing so algorithmically fairly simple. The algorithm from [SKS11] below, with some modifications for clarification, will ensure that the lock and unlock instructions are created:

1. Scan the transaction from the beginning moving one statement at a time.
2. If the next instruction is a read of an item i and there is no lock on i already, insert an instruction to lock i

in shared mode before the read.

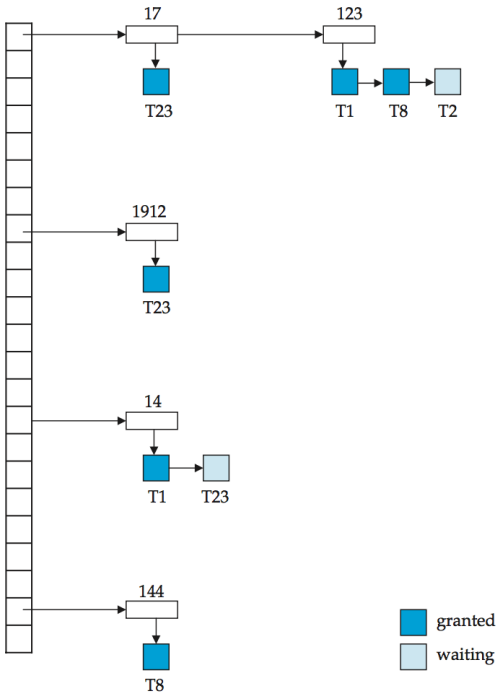
3. Otherwise, if the next instruction is a write of an item i and the item i is not already locked exclusively:
   (a) If the item is locked in shared mode, insert an instruction to upgrade the lock on i to exclusive mode before the write.
   (b) Otherwise the item is not locked, insert an instruction to lock i in exclusive mode before the write.
4. At the end of the transaction (commit or abort), insert instructions to unlock all locks that were locked (regardless of mode).

Following this systematic approach will ensure that locks are shared when they can be, never permit intermediate state to be visible, and ensure that transactions can be serialized.

Implementation of the Lock Manager. The lock manager is the mysterious part of the database that decides when to grant lock requests, and when to deny them. In fact, the lock manager receives the requests to lock and unlock commands. The lock manager may return a message granting the lock when the request is received, or a denial message. But that answer might not come right away: if a grant cannot be completed right now we can delay it until it can be granted. Rejection, however, is fatal to the requester: a rejection means the requesting transaction must be rolled back. The unlock requests do not strictly speaking require a response, but it is noteworthy for the lock manager since it allows other transactions to be unblocked.

We will cover two approaches: lock tables (a linked-list approach) and a graph algorithm.

The linked list approach means that each data element is associated with a linked list. In that linked list is the current transaction owning that lock as well as any other transactions that are waiting for that particular item. The image below shows a lock table example from [SKS11] in which there are locks for five different data items numbered 14, 17, 123, 144, and 1912:



The rules for constructing and manipulating such a lock table are pretty simple and comprehensible from the diagram. Each (lockable) data item is associated with a linked list. Entries in that linked list contain the transaction identifier and an flag to indicate granted or waiting.

If a request arrives for a data item for which there is no lock, then an entry is created and the request can be immediately granted. If a lock already exists for that data item, a new entry is created. As for whether it is

granted right away or not, compatibility must be checked. If it is compatible it may be granted; otherwise it must wait.

When there is an unlock request, the entry for that transaction and data item are removed and, if possible, further requests are granted. If a transaction aborts, any locks (granted or not) are removed from the lock table.

This algorithm avoid the possibility of starvation, because the linked list imposes order on the request such that later requests always are enqueued after earlier ones [SKS11].

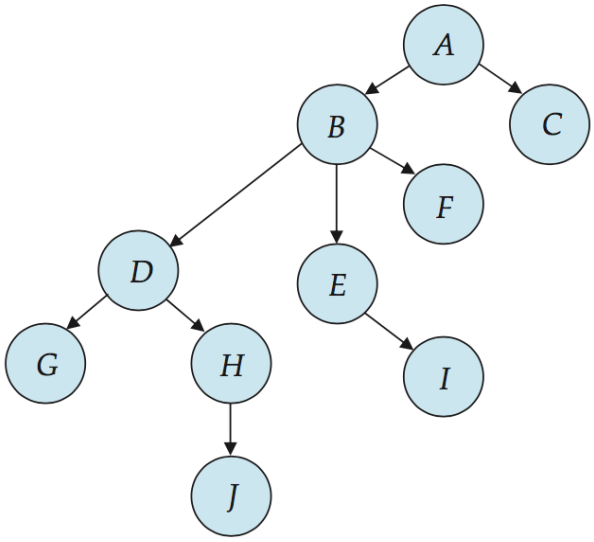
The graph based protocol in [SKS11] has to do with establishing a lock based protocol that uses a tree. To make this work, all data elements need to be assigned an identifier (an ID of some sort) which allows establishment of partial ordering on the data elements. The purpose is so that it is possible to enforce the rule that they are acquired in order.

You might recall this sort of protocol from the general discussion of deadlock in a concurrency context. In the examples, a simple deadlock is created when one thread locks `a` and then `b` and the other thread locks `b` and then `a`. The solution was to establish an ordering on the lockable elements (in that case, the alphabetical ordering seemed logical enough, but reverse alphabetical would work too). That works okay for names, but not everything has a name, and anything relatively simple like memory location of file location would work to impose an ordering.

The tree protocol is an example in [SKS11] that works only for exclusive locks with four rules:

1. The first lock by a transaction T_1 may be on any data item.
2. Any other data item d may be locked by T_i only if the parent of d is currently locked by T_i.
3. Items may be unlocked at any time.
4. A data item that has been locked and then unlocked by T_i cannot later be locked again by T_i.

The diagram below illustrates a potential ordering of data elements:



Tree structured ordering of data elements [SKS11]

It is simple enough, when looking at this graph, to think of some schedules that are legal or illegal under this protocol. It can be shown that this protocol ensures freedom from deadlock. The reasoning is a fairly simple proof by contradiction: assume that a deadlock exists and then show a contradiction.

In [SGG13] is a proof that ordering the resources prevents deadlock in processes, and it is simple enough to modify for transactions. Let us say there is a function $f()$ that takes a data element and returns a unique identifier used

for ordering. Assume a circular wait is present. Let the set of processes in the circular wait be $\{T_0, T_1, ...T_n\}$ and the set of resources (data elements locked) be $\{R_0, R_1, ...R_n\}$. The cycle is formed as: T_i is waiting for resource R_i and that resource is held by T_{i+1}. The exception is the case of T_n, which is waiting for resource R_n that is held by T_0 (completing the cycle by wrapping around). Since Process T_{i+1} holds resource R_i while requesting R_{i+1}, this means $f(R_i) < f(R_{i+1})$ for all i. But this means that $f(R_0) < f(R_1) < ... < f(R_n) < f(R_0)$. It cannot be the case that $f(R_0) < f(R_0)$: a contradiction, meaning a circular wait cannot occur.

Use of the tree structured protocol does not, however, ensure recoverability or prevent cascades from occurring [SKS11]. That means we need to also take steps, as earlier, to ensure that the schedule conforms to our other properties. The other negative here is that the database server might need to lock items other than the ones strictly necessary in the transaction.

Consider an example in the tree lock diagram above; if data elements A and J are needed, then items B, D, and H also need to be locked to make this work. Moreover, it is very likely that we are going to lock the root of the tree, meaning that (most) transactions are effectively serialized. This is probably not a good thing.

The tree protocol is useful if the goal is to prevent deadlock from happening, but it is not a very good solution given the tradeoffs. It is more likely that we will just allow transactions to proceed and detect if they do get stuck, and if so, try to do something to fix it.

25 — Deadlock, Lock Granularity

Deadlock

You should recall from learning about concurrency what deadlock is about, and therefore a brief review only is in order. Those in need of a refresher are advised to take a look through previous courses' material to get caught up. Others may find this to be review material, but it does deviate in some significant ways from the discussion of deadlocked processes.

We have already introduced the subject of deadlock and gave an informal definition as all transactions being "stuck" (unable to proceed). A more formal definition is given in [Sta14] (but modified for a database): "the *permanent* blocking of a set of transactions that either compete for system resources or communicate with each other". There is emphasis on permanent. It may be possible for all transactions to be stuck temporarily, because one is waiting for some event (e.g., a read from disk), but this situation will resolve itself and is not deadlock. A set of transactions is truly deadlocked when each transaction in the set is blocked on some event that can only be triggered by another blocked transaction in the set. In this case it is permanent, because none of the events can take place.

Remember, there are four conditions necessary for a deadlock to take place:

1. **Mutual Exclusion**: A resource belongs to, at most, one transaction at a time.
2. **Hold-and-Wait**: A transaction that is currently holding some resources may request additional resources and may be forced to wait for them.
3. **No Preemption**: A resource cannot be "taken" from the transaction that holds it; only the transaction currently holding that resource may release it.
4. **Circular-Wait**: A cycle in the resource allocation graph.

We could choose not to handle deadlock. This option is certainly convenient for database system designers: we simply pretend that deadlock can never happen, or if it does happen, it is the application developers' fault. While it is tempting and easy to just define a problem as being nothing we need to deal with, that's unrealistic in reality. If two transactions get deadlocked, the user may simply see no progress and just feel bad about it. But sad users are not a good thing, now, are they?

Although in theory it would be nice to prevent deadlock from ever occurring, in practice it is probably more likely that we will simply try to detect it and deal with it if it occurs. But prevention is worth looking at.

Deadlock Prevention

The first three conditions for deadlock (mutual exclusion, hold and wait, and no preemption) are all necessary for deadlock to be possible. If we eliminate one of these three pillars, deadlock is not possible and it is prevented from happening. If we prevent the circular wait from occurring, we are still in a world where deadlock can occur but we cleverly avoid it taking place based on how we allow locks to occur.

Mutual Exclusion. This pillar cannot, generally speaking, be disallowed. The purpose of mutual exclusion is to prevent errors like inconsistent state or crashes. Getting rid of mutual exclusion to rule out the possibility of deadlock is a cure that is worse than the disease. It is therefore not acceptable as a solution.

Hold and Wait. To prevent the hold-and-wait condition, we must guarantee that when a transaction requests a resource, it does not have any other resource. One plausible solution is that the transaction must request all resources at the beginning of the program. So if the transaction is going to need resources R_1, R_2, and R_3 at some point in the program, all three must be requested right at the beginning and held throughout the transaction. No further resources may be requested at any time during execution [SGG13].

When talking about processes, actually we said this requires some amount of clairvoyance: a process has to know in advance all of the resources that it will need. Remember that a file is a resource, and even a simple program like a text editor can be used by the user to open an arbitrary file; how do we know in advance which will be requested? But with database transactions, actually, we can actually evaluate what data items are needed. That might not be trivial, but at least for databases it is possible.

This also has performance implications: a transaction cannot start until it has all the resources it will ever need, even if it will not need them until much later. Thus, transactions might spend a lot of time waiting before starting; something that users may not tolerate. In theory, a transaction might never start if one or more of the resources it needs is always in use (so this is vulnerable to starvation).

No Preemption. If we violate this condition, it means that we do have preemption: forcible removal of resources from a transaction. In the database context, if we have to preempt a transaction, it means the transaction in question is rolled back and then restarted. To work out an ordering, transactions are assigned a timestamp. If a transaction is rolled back, its timestamp remains the same [SKS11]. There are two approaches for what happens if we need to do pre-emption:

The first is called *wait-die*: if a transaction T_i requests an item held by T_j, then the timestamps of these two transactions are compared. T_i will be allowed to wait if it is older (i.e. its timestamp is a smaller number) than T_j, otherwise it "dies" (is rolled back).

The second is *wound-wait*: if a transaction T_i requests an item held by T_j, then the timestamps of these two transactions are compared. If T_i is the younger transaction (its timestamp is larger), then T_i can wait. Otherwise, T_j is "wounded" by T_i and T_j is rolled back.

So, in case of a conflict, one of the two transactions is going to be rolled back. And it is always the younger one. This is not because older transactions vote in higher numbers, but because it tends to be more expensive to restart an older transaction. A transaction that has been running for a long time, if it is restarted, has to do a lot more work to get to the point where it was terminated than a younger transaction. Another reason: if the oldest transaction were constantly the one selected, that transaction itself might never get to run to completion (starvation) because it is constantly killed before it finishes. A third reason is somewhat more subtle: if the killing transaction is very aggressive then perhaps no tasks run to completion because each transaction, shortly after becoming the oldest, is claimed by the deadlock recovery transaction. Therefore, young transactions tend to be the ones selected.

In either approach there may be unnecessary rollbacks [SKS11].

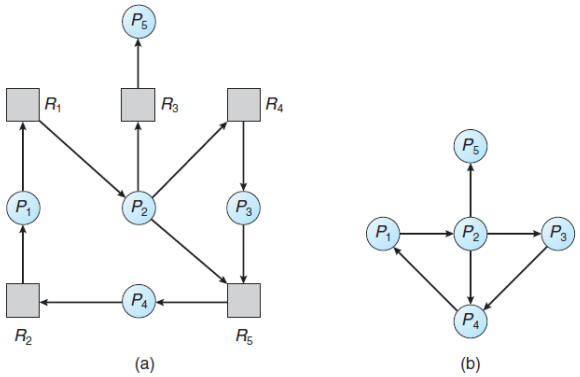
Instead we could also have locks that have timeouts: when a lock is requested there is a defined maximum time the transaction is willing to wait. If that time limit is reached, the transaction rolls back automatically and begins again. The obvious difficulty lies in choosing the length of time that the transactions will wait: too long and a deadlock persists for a lengthy period of time; too short and transactions do not complete because they are rolled back repeatedly. And that is generally unfortunate. This sort of approach is not especially recommended.

Circular Wait. We have already discussed the idea of using two phase locking to put an ordering on the locks and we need not repeat that yet again.

Deadlock Detection

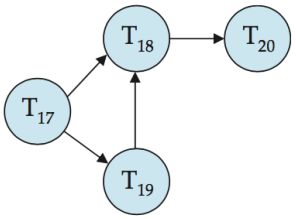
In previous courses you may have covered algorithms like the banker's algorithm and the graph algorithm for detecting deadlock. It is the second one that is interesting in the database.

A quick review of the graph algorithm follows, then. If resources have only a single instance, we may reduce the graph to a simplified version called the *wait-for* graph. This removes the resource boxes from the diagram and indicates that a process P_i is waiting for process P_j rather than for a resource R_k that happens to be held by P_j. An edge $P_i \rightarrow P_j$ exists in the wait-for graph if and only if the resource allocation graph has a request $P_i \rightarrow R_k$ and an assignment edge $R_k \rightarrow P_j$ [SGG13]. Consider the example below:



(a) A resource allocation graph and (b) its corresponding wait-for graph [SGG13].

Because locks are unique (there is always only one instance) then there is no need to use the resource allocation graph and we can always use the simplified wait-for graph for transactions as below:



Transaction wait-for graph [SKS11].

Given the wait-for graph, it is trivial for humans to look at this and determine if there is a cycle, but for the computer it takes slightly more work. We must execute an algorithm to determine if there is a cycle. A cycle exists in the wait for graph if and only if a deadlock exists in the system. Such cycle detection algorithms tend to have runtime characteristics of $\Theta(n^2)$ where n is the number of nodes in the graph. Though not a formal proof and probably not acceptable to write on a data structures and algorithms examination, the premise of the algorithm is: for each node n in the graph, examine each possible path from that node. If a node is reached from which no further path is available, examine the next path. If node n is reached on the current path, a cycle is detected and the algorithm terminates.

The runtime characteristic of the simple deadlock detection algorithm was identified as $\Theta(n^2)$. This means that the deadlock detection routine is expensive to execute.

This prompts a question: how often should the deadlock detection algorithm be run? One strategy is to run it every time a resource is requested. Running the algorithm might be rather expensive, so perhaps this is too often. An obvious optimization: it should only run every time a resource request cannot be granted (that is, a transaction is waiting). Another idea: run it periodically instead.

When to run the deadlock detection algorithm depends on how often we expect deadlock to occur, and how severe a problem it is when deadlock occurs. If deadlock happens a lot, checking for deadlock often will make sense. If

the consequences of a deadlock are severe, it makes sense to check frequently to identify the problem as soon as possible.

Deadlock Recovery

It is clear that our recovery strategy in the database context is rollback. But rather than just simply rolling transactions back and restarting them.

On the subject of what transaction to select as a victim we already identified the timestamp as one possible way to decide, but there exist a number of other possible criteria [SKS11]:

1. How long the transaction has been executing.
2. How long is remaining in execution (see the estimates from the execution plan).
3. What resources the transaction has used.
4. Future resource requests, if known.
5. How many times, if any, the transaction has been selected as a victim (to prevent starvation).

After a victim has been selected, we might reason, it's pretty easy from here: simply roll that transaction back. That would be a total rollback: undo all the steps of the transaction and start from the beginning once again. There exists, however, the ability to do a *partial rollback*.

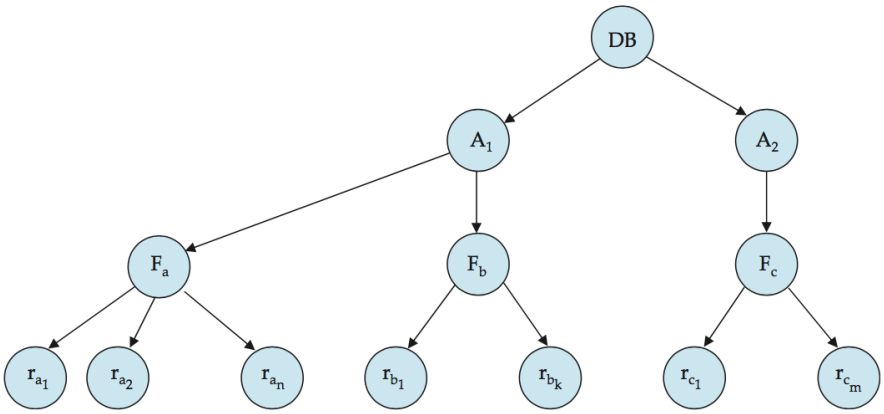
The goal of the partial rollback is to rollback the transaction back only as much as is necessary to break the deadlock; this is accomplished by maintaining the sequence of lock requests and grants. By finding the lock(s) that are implicated in the deadlock, we can roll back a transaction to where it obtained the first of such locks [SKS11]. This can be beneficial if the amount of work saved is large enough to outweigh the effort required to determine where the lock conflict occurs. For small transactions, a full restart is probably sensible; for longer ones it might be worth it to try to do a partial rollback instead.

Lock Granularity

We have choices about the granularity of locking, and it is a trade-off (like always). Locks' extents constitute their *granularity*: that is, how much data is being protected by such a lock.

Coarse-grained locking is easier to implement, but it can significantly reduce opportunities for parallelism. *Fine-grained locking* requires more careful design, increases locking overhead and is more prone to bugs (deadlock etc). In coarse-grained locking, you lock large sections of your data with a big lock; in fine-grained locking, you divide the locks and protect smaller sections with multiple smaller locks.

In the database we could define multiple levels of granularity. As in the diagram below:



Granularity hierarchy showing four levels [SKS11]

In this diagram, there are four levels of the hierarchy. At the lowest level, individual records may be lockable. Above that, it may be at file level (i.e., relation level). The third level from the bottom is an "area" – this is usually a grouping of one or more relations that form a logical group. The root of the tree is the database itself: the entire database can be locked if needed. A lock of a node in this hierarchy also implicitly locks all descendants of that node.

This leads to a fair amount of headaches: if a transaction T_i wants to lock a node x then all the children of x must be examined to see if they are locked. Also, we have to look to see if any of the nodes between the root and x are locked (including the root). And this applies every time an item is to be locked! This is painful.

The more efficient route recommended by [SKS11] is called *intention lock modes*, which are put on the ancestors of a node before the node is locked. These intention locks can be checked while traversing the tree. The following modes are possible:

- **Shared** (S) - This is as before, directly locking an item in shared mode.
- **Exclusive** (X) - Directly locking an item in exclusive mode.
- **Intention Shared** (IS) - There is at least one shared lock at a lower level of the tree, but no exclusive locks.
- **Intention Exclusive** (IX) - There is least one exclusive lock at a lower level of tree, and there can be shared ones as well.
- **Shared Intention Exclusive** (SIX) - The current node is locked in shared mode, and there is at least one exclusive lock at a lower level (and maybe shared ones too).

Then a transaction must follow the six rules below [SKS11]:

1. All attempts to lock must observe the table (below) as to whether they are permitted to proceed (or wait).
2. The transaction must lock the root of the tree first (in any mode, but it must pick one).
3. A node n can be locked in **S** or **IS** mode only if the parent of n is locked in either **IX** or **IS** mode.
4. A node n can be locked in **X**, **SIX**, or **IX** mode only if the parent of n is locked in either **IX** or **SIX** mode.
5. A node n may only be locked if the transaction has not previously unlocked any nodes.
6. A node n may only be unlocked if none of the children of n are locked by this transaction.

Compatibility	**IS**	**IX**	**S**	**SIX**	**X**
IS	✓	✓	✓	✓	×
IX	✓	✓	×	×	×
S	✓	×	✓	×	×
SIX	✓	×	×	×	×
X	×	×	×	×	×

It is noteworthy that locks must be acquired in a top-down manner and then released in a bottom-up manner.

26 — Timestamp, Validation, & Multiversion Protocols

Concurrency via Timestamps

Beyond just doing concurrency through some sort of locking protocol, we can also use timestamps for arranging concurrency. The timestamp, as you will recall, is a unique identifier, and they are typically assigned in the order the transactions are submitted to the system [EN11]. If timestamps are used appropriately, then consistent results are produced without the use of any locks or locking, meaning deadlock can never occur.

The notation for the timestamp of a transaction T is $TS(T)$. The transactions can be ordered based on their timestamps; if a transaction T_i arrives and then later another transaction T_j arrives, then $TS(T_i) < TS(T_j)$. If two transactions arrive at exactly the same time, then some sort of serialization procedure will be needed to make sure that they are not identical. It is commonly the case that the system clock is used to get the time stamp; just read the current value of the clock and use that.

The alternative is a logical counter: increment a simple integer counter for each transaction. After a timestamp is assigned, just increment the counter. Eventually there is a limit (even if you can have 2^{64} transactions, in theory, which is quite a lot, but not infinite) but at some point the transaction counter will need to roll over or reset.

Generation of the timestamps is fairly simple, it would seem, but what are they for? Timestamps are used for serializability order in the schedule; the system must ensure that the schedule executed is equivalent to a serial schedule in which transactions are ordered according to their timestamps (ascending) [SKS11]. Note that this is very different from two-phase locking. In two phase locking, a schedule is serializable by being equivalent to some schedule that is permitted under the locking rules; in timestamp ordering the schedule is equivalent to the a particular serial ordering matching ascending transaction timestamps [EN11].

Every data element in the database is associated with separate timestamps for reading and writing. The notation can vary: in [EN11] they are called **read_TS** and **write_TS**; in [SKS11] they are **r-timestamp** and **w-timestamp**. To save space I might personally like TS_r and TS_w. Each time the a data element X is read or written, its appropriate timestamp is updated to the timestamp of the transaction performing the read or write.

The formal definition of the read timestamp of an item X: the largest timestamp amongst all the timestamps of transactions that have successfully read X. So $TS(X) = TS(T_k)$ where T_k is the youngest transaction that has read X successfully [EN11].

The formal definition of the write timestamp of an item X: the largest timestamp amongst all the timestamps of transactions that have successfully written X. So $TS(X) = TS(T_k)$ where T_k is the youngest transaction that has written X successfully [EN11].

Unfortunately, simple timestamp ordering does not avoid the risk of cascading rollback. Schedules are not guaranteed to be recoverable. This leads us to the *timestamp ordering protocol* as described in [SKS11].

For a read operation T_i is performing on X:

1. If $TS(T_i) < TS_w(X)$ then T_i needs to read a value of X that was already overwritten – and therefore the

read operation is not permitted and T_i is rolled back.

2. If $TS(T_i) \geq TS_w(X)$ the read operation is executed and TS_r is set to $\max(TS(T_i),\ TS_r(X))$.

For a write operation T_i is performing on X:

1. If $TS(T_i) < TS_r(X)$ then the value that T_i is producing was needed previously but the system assumed it wasn't going to happen and proceeded without it; thus the write is not permitted and T_i is rolled back.
2. If $TS(T_i) < TS_w(X)$ then T_i is attempting to write an obsolete value; the write is not permitted and T_i is rolled back.
3. Otherwise, the write proceeds and $TS_w(X)$ is updated to $TS(T_i)$.

When a transaction is rolled back and restarted, the transaction is assigned a new timestamp. The timestamp ordering protocol does provide us with conflict serializability, because conflicting operations are processed in timestamp order. Because there are no locks, there cannot be deadlocks – but can starvation still occur? The answer is yes, a very long transaction might be constantly frustrated by shorter transactions constantly swooping in and making changes that force a rollback of the long transaction. Unfortunately it might be necessary to block other transactions and let the long one finish [SKS11].

In the example below, transaction T_{25} reads values A and B , and transaction T_{26} modified both A and B:

T_{25}	T_{26}
read(B)	
	read(B)
	$B := B - 50$
	write(B)
read(A)	
	read(A)
display($A + B$)	
	$A := A + 50$
	write(A)
	display($A + B$)

A schedule produced with timestamp ordering [SKS11].

It is worth noting that neither two phase locking nor the simple timestamp ordering protocol covers all serializable schedules; that is, there are some that are valid under one but not valid under the other [EN11]. That's fine, as far as we are concerned – we don't need to consider all possibilities; we will get enough choices.

If we would like to have recoverability we can use strict timestamp ordering. In strict timestamp ordering, a transaction T_i that issues a read or write of an item X where $TS(T_i) > TS_w(X)$, then the read or write operation is delayed until the transaction that last wrote X has either committed or aborted. This, in some way, requires simulating locking item X until a previous write has been committed or aborted, but there cannot be a deadlock because a transaction can only wait on an older transaction (one with a lower timestamp) [EN11].

Thomas's Write Rule. A modification of the basic algorithm called Thomas's Write Rule (or the Thomas[20] Write Rule) allows greater concurrency than the basic algorithm and also, hopefully, rejects fewer writes. To explain it simply, it is "ignore outdated writes".

More formally, the write item checks are modified to the following [EN11]:

1. If $TS_r(X) > TS(T_i)$ then abort and roll back T_i (write rejected).
2. If $TS_w(X) > TS(T_i)$, then do not execute the write, but continue executing (do not roll back the transaction).

[20]Named after the author of the paper describing this rule, Robert H. Thomas

3. Otherwise, the write proceeds and $TS_w(X)$ is updated to $TS(T_i)$.

This requires some explanation. If a transaction T_i is going to write an old value, under the previous scheme that write would be rejected. Under Thomas's Write Rule, we will just skip doing the write (the value to be written is outdated) and we just carry on. Essentially we just pretend that the write happened but was immediately overwritten by the more up to date value.

Validation-Based Protocols

As you know, a transaction that is read-only does not interfere with any other read-only transaction. If a database has many more reads than writes, then a concurrency control routine might do a lot of really unnecessary work and maybe delay transactions unnecessarily. If we could instead just step in where transactions may be in conflict, that would be better, but it is necessary then to find out what transactions are the ones that have the potential for conflict... [SKS11].

The validation protocol means that transactions have three phases (although read only transactions skip the last one). The phases are described in [SKS11] as:

1. **Read Phase.** The transaction is executed, reading the value of items and storing them in temporary variables local to that transaction. Writes are also done on temporary local variables without any changes to stored data.
2. **Validation Phase.** A validation test is applied (described below). If the transaction passes the test, it may proceed to step 3; if it fails, then the transaction is aborted.
3. **Write Phase.** The temporary local variables are written out. Obviously, read-only transactions have nothing to do in this phase and may skip it altogether.

The validation test for the current transaction T_i, for each transaction T_k where $TS(T_k) < TS(T_i)$, one of the following conditions must hold:

1. T_k finishes before the start of T_i: serializability order is maintained because the current transaction starts after the previous one finished.
2. The set of data items written by T_k does not intersect with the set of data items read by T_i, and T_k finishes writing before T_i starts its validation phase. This ensures that the writes of these transactions do not overlap.

To make this work out as expected we need, now, three timestamps for each transaction: one for the start of the transaction, one for the start of the validation phase, and then a final timestamp for when the transaction is finished.

Validation prevents cascading rollbacks because the writes don't really take place until the validation phase has taken place. This is called *optimistic concurrency control* because transactions execute optimistically, assuming that they are going to finish (and we'll sort it out if something goes wrong) [SKS11]. The locking and timestamp approaches are a bit more pessimistic because they force a rollback if there is a potential for a conflict. The potential for a conflict is not a guarantee that one will occur.

That potentially leads to an interesting discussion on the subject of whether an optimistic or pessimistic type approach is better. The general recommendation is that optimistic approaches are lower overhead and better for situations where we expect there is not much contention. We get more concurrency out of optimistic schemes, which would theoretically mean better performance.

A small example to demonstrate the validation routine:

T_{25}	T_{26}
read(B)	
	read(B)
	$B := B - 50$
	read(A)
	$A := A + 50$
read(A)	
< validate>	
display($A + B$)	
	< validate>
	write(B)
	write(A)

A schedule produced in a validation scheme [SKS11].

Multiversion Schemes

Thus far we have said that a transaction must be either delayed or denied (aborted) to ensure serializability. In a multiversion scheme, we have multiple versions of the data at the same time; instead of rejecting a read because a write has taken place in the meantime, the read is provided an older version of the item. Choosing which one is meant to be read is the hard part of this routine, of course.

This does require more storage space to maintain multiple versions of the data. Old versions might be maintained anyway for recovery, or just record keeping purposes [EN11]. Many databases are used in environments where all changes must be tracked for legal reasons, with data like the changer, change time, and what the data changed was. If that is the case then there is no additional cost to storing the data since the older versions are already present regardless.

If that is not the case, then yes, multiple versions take extra space. The amount of space is determined by how many data items are modified, in a given period of time and how long we would like to maintain older versions. More writes means more space needed. And the longer we maintain them, the more space is needed.

Timestamp Ordering. In the multi version technique using timestamps, for a data element X, there are multiple versions $X_1, X_2, X_3, ..., X_k$. For each X_i there are read and write timestamps. The read timestamp is the largest timestamp that has successfully read X_i; the write timestamp is the timestamp of the transaction that wrote the value of this version of X [EN11].

Whenever a transaction is allowed to update a value, a new version of X is created. As it has never been read, yet, the read timestamp should be the same as the write timestamp (until it is read of course). If a read takes place, then the read timestamp is updated if the transaction doing the read has a larger value than the current read timestamp.

Ensuring serializability requires holding to just two rules [EN11]:

1. If transaction T tries to write item X, and version i of X has the highest write timestamp of all versions of X that is also less than or equal to the timestamp of T, and the read timestamp of X_i is larger than the timestamp of T, the abort the transaction T and roll it back. Otherwise, create a new version of X with read and write timestamps equal to the timestamp of T.
2. If transaction T wants to read item X, find the version of X (X_i) that has the highest write timestamp of all versions of X that is also less than or equal to the timestamp of T. Update the read timestamp of X_i to the maximum of its current read timestamp and the timestamp of T.

The goal is that read operations are always successful, even if they read an older version. A write, however, might not be permitted if it comes "too late" – trying to write a version that another transaction would have read. The drawback is that every read requires us to update the read timestamp, which is itself a write operation (and may result in a disk access).

When do we know that we can get rid of an old version? An old version is no longer needed when its write timestamp is younger than that of the oldest transaction in the system. But one must be careful then to only delete old versions; at least the current version must be maintained. But also it means that in a period in which there are no transactions active, we may clean up any old versions. That may not occur frequently enough in a heavily used database, so it might be a periodic task to simply trash very old versions.

A major drawback of this scheme is that we resolve any conflicts on writes through rollbacks, which are expensive, rather than through waiting (which is unpleasant but tolerable) [SKS11]. Then instead, an alternative:

Multiversion Two Phase Locking. What if we said we want an all-of-the-above solution that combines two phase locking with the multiversion scheme? This is that solution. The advantage of multiversion schemes is that reads don't fail. The advantage of two phase locking is that locks allow transactions to wait rather than roll back, and deadlock is avoided. The plan is thus to allow each transaction to be handled in the way they like best. Thus, read-only transactions are differentiated from the update transactions.

Update transactions use two phase locking, with the restriction that they hold locks up until the end of the transaction, meaning that there is only one timestamp for the transaction, which is used to both serialize the transactions and to assign the version of the data written [SKS11]. The timestamp under this scheme is an incrementing counter rather than the actual time of the transaction.

Read-only transactions are assigned a timestamp based on the current value of the counter for update transactions, at the time when the read transaction is created. Multiple read transactions can have the same value of their timestamp. In effect, the read transaction's timestamp k is declaring that its view of the data is the version at time k. A transaction that reads some data X with its timestamp k gets the version of X with the largest timestamp less than or equal to k.

If an update transaction wants to read an item, it acquires a shared lock on the latest version of the item; to do an update it acquires an exclusive lock and creates a new version of that item with the timestamp being the maximum value (effectively: infinity) [SKS11]. The new version has this infinite timestamp so it won't be read by any read transaction, but it will be overwritten when the write transaction commits: the commit operation sets the timestamp on all newly-modified variables to be the value of the transaction performing the update plus one. Then the timestamp counter is incremented. This does mean that only one transaction can commit at a time.

27 — Snapshot Isolation, Weak Consistency, Insert/Delete

Snapshot Isolation for Concurrency Control

We already got a look at the idea of transaction isolation but we are now going to examine more carefully how it works behind the scenes. In general, the idea is that every transaction has its own "world" it can operate in and then we need to merge the result when the transaction is ready to commit. And that needs to be done atomically.

We can use validation to decide whether a transaction is allowed to commit. This applies really only for transactions that do at least one write. Reads don't interfere with one another, really, and if the two writes are on disjoint items, there are no problems there either. It only gets interesting if there are two transactions that have items in common that run at the same time. Remember that here, when we talk about concurrency, we mean two transactions that are "active" at the same time.

Usually, we do not like to just allow both transactions to go forward because the first write is then overwritten by the second; a "lost update" [SKS11]. This is sometimes acceptable as a choice, even if some textbooks say that this is really undesirable and horrible.

The first snapshot isolation strategy is called *first committer wins*: whichever transaction T is ready to commit first has to pass a simple test. If any transaction concurrent with T has already written an update to any data item that T wants to write, T is rolled back; otherwise T commits and its updates are written to the database. It's called first committer wins, because whatever transaction gets to the commit statement proceeds and any later transactions are rolled back.

The alternative is *first updater wins* and this is pretty much exactly what it sounds like: it is the timing of the write rather than the timing of the commit that indicates which transaction is allowed to proceed and which one(s) must be rolled back.

Locks are always released when the transaction either commits or aborts as is necessary. But strangely, the fact that a transaction T_2 writes a value later than T_1 writes that same value might succeed if for some other reason the first transaction aborts [SKS11]. But to keep things from getting rather confusing it is likely a simple implementation will just force the abort if T_2 finds it wants to get a write lock on a data item that is currently exclusively held. Failure equals death indeed.

This scheme as presented, however, does not ensure serializability. There are two suggested scenarios for how this might happen [SKS11]:

Example 1: We have two concurrent transactions T_1 and T_2 that operate on A and B. If T_1 reads A and B, then updates B and T_2 reads A and B and updates A, we have the potential for a conflict. Neither transaction sees the updates of the other when they do their reads, when it comes to the write... well... neither one has a problem, strictly speaking, because they write different data items and whether first-updater-wins or first-committer-wins takes place, neither transaction is detected as conflicting and neither is rolled back.

In spite of that, we have a problem: we have an outcome (output data) that could not happen if we had serial execution of the transactions; either T_1 then T_2 or T_2 then T_1. Now what?

It looks like under this scheme, some things like foreign key checks, primary key checks, et cetera, cannot be checked in the transaction world itself and must be checked an extra time when the transaction commits and is finally ready to replace the actual stored version on the database.

Example 2: Imagine we still have T_1 and T_2 and A and B as before. T_1 reads B and updates B, and T_2 reads A and B and updates A. As before, there are no direct conflicts on the data items so neither first-committer-wins nor first-updater-wins would detect a problem. Furthermore, there's a possible serial order here: there are no conflicts on A, and no cycle in the precedence graph: T_2 reads the value of B that existed before the write by T_1.

A third transaction, even one that is read-only, can cause the cycle to occur, however. If T_1 commits and T_2 is still running, and T_3 is created and its view of the world has the update from T_1 but not T_2 and suddenly we have the cycle in the precedence graph, rendering the schedule non-serializable.

After all that, though, we can accept some degree of non-serializability as long as it does not produce inconsistent results. For performance and other reasons it might be sensible to just let execution orders that are not serializable proceed as long as they don't produce inconsistent results. Or we will allow such things if we are convinced that inconsistencies don't make a big difference.

A small example of when we might have to really enforce constraints is the primary key. If you have a sequential counter of some sort, an ID number, then two newly created entries in the database would each be looking at the old snapshot data and will produce the same new number. Neither transaction can detect that another has tried to use the same ID number, but only when we are actually ready to perform the second commit will the duplicate key value be detected.

This is, actually, slightly different from what we have discussed thus far, because it focuses on an insertion rather than just a simple write. That is different...

Changes: Insertion & Deletion

If you can believe it, all the discussion thus far on the subject of concurrency control has excluded the possibility of performing an insertion or deletion on the data items; only read and writes were allowed. But those are pretty important operations, so we can't ignore them forever.

Deletion. The impact of delete depends, obviously, on what the else is going on at the time. We can take a view of what happens by considering the relation of the delete instruction relative to other instructions. In each of the cases, instruction I_1 is our delete instruction [SKS11]:

- **Read instruction**: If the delete $I_1(X)$ happens before a read of X then the read will get a logical error; if the read happens before the delete, there is no problem.
- **Write Instruction**: If the delete happens before the write, the write will get a logical error; if the write happens before the delete, there is no problem.
- **Delete Instruction**: Whichever instruction goes first will succeed; the second one will have an error (can't delete a deleted item).
- **Insert Instruction**: If the element doesn't exist to begin with, the delete can't be before the insert, otherwise there's nothing to delete... If the item did exist then a delete should go first so the new item replaces it.

It is sensible to conclude that the delete operation requires an exclusive lock on that data item before it can be deleted. Also, the rules for writing apply for a deletion action if using a timestamp ordering protocol. It is probably not a stretch to imagine it is very much like a write.

Insertion. The insert operation conflicts were explained above when discussing deletion, so they don't need to be repeated here. Insert is also a lot like a write, so again, the insert operation is treated like a write: the newly created item is treated as exclusively locked by its creating transaction until the transaction commits [SKS11].

I think it bears repeating at this level that the changes we want to make here are at the level of actual manipulation of the underlying data. It is perfectly valid, from the perspective of the user writing the SQL, to do a read and get back "no results" because the item was very recently deleted. The problem here is that we have identified what data elements we need to access and the ones we want are not there (or not as expected, or there are extra ones...) and we need to stop, go back, reformulate the list of things to do, and then try to do it again.

The Phantom of the Op... Database. Suppose the following transactions are executed concurrently: (1) a select statement that finds all books published by author "Jim Butcher"; (2) an insert statement that adds a new book with author "Jim Butcher" amongst its attributes.

There are two possibilities for how this might go: T_1 happens before T_2 or vice versa. If T_2 happens first, nothing weird happens – in the serial schedule, first it's T_2 then T_1 and the new book appears in the results. In the second case, if T_1 goes first and it does not contain the new book, then in the serial schedule, T_1 appears first.... except... these transactions do not operate on any tuples in common and yet there is still some dependencies between them, and they conflict on a "phantom" tuple (one that doesn't exist) [SKS11]. This is called the *Phantom Phenomenon* and has nothing at all to do with the breakout success of the Andrew Lloyd Webber musical.

The phantom phenomenon can also happen if a read changes the value, say, if an item was entered incorrectly as "Jim Butchr", a corrective update statement run at the same time as T_1 above produces the same problem.

As a first solution, you might consider that an insertion or deletion should exclusively lock the relation entirely. That's one way to do it, it's certainly likely to be effective at getting rid of the phantom problem, but it is a painful solution because it forces a lot of transactions to wait or roll back when we really don't need that. The goal is to prevent insertions and updates that match the predicate in question, but only those (keeping the scope of the lock as minimal as possible).

Remember the index? So far we have just assumed that data items being updated are tuples, but that's not strictly true, because index data elements are also data and need to be lockable. Finding items that match a search predicate requires searching an index or performing a full table scan to find the data [SKS11].

Idea: let's associate a data item (available to be locked) with each table, with the goal that it represents the way to find tuples in the relation [SKS11]. Any time we want to update the data about what is in the relation, and transactions that want to read what tuples are in the relation need to also lock this item (in shared mode perhaps). And anything that operates on an index must lock the index. This moves the lock out of the realm of shadows and into the "real" world – the lock conflicts here are now on an actual lock item.

Keep in mind this is very much distinct from locking the entire table: locking is still required on tuples and holding this sort of lock only restricts whether other transaction can update (or read) the information about what tuples are in the relation [SKS11]. Even so, it's not all that dissimilar from the idea of having one lock for the whole table, in that it almost serializes all transactions.

An alternative is *index locking*, which allows us to lock parts of the index at finer granularity. The protocol is [SKS11]:

- Every relation must have at least one index.
- A transaction T_i can only access tuples after it has found them using an index (if we must do a full table scan, it's treated as if we just crawled through the whole of some index...)
- A transaction that performs a lookup (finding one tuple or multiple) must get a shared lock on all the index leaf nodes it accesses.
- A transaction may not insert, delete, or update any tuples in the relation without updating every index of that relation. That requires getting exclusive locks on all index leaf nodes affected by the operation.
- Locks are still obtained on tuples as usual.
- Two phase locking protocol still must be followed.

Predicate Locking. You may imagine that the idea of index locking does not actually match very well with the idea of making the locking match the predicates only, such as only checking on things that affect author. That sort of locking technique sounds good but it is more difficult to implement.

Weak Consistency

The SQL standard provides several isolation levels, and it's not necessary that we stick with serializability as the level of consistency we are willing to accept. We can weaken the rules a bit to get some more performance out of the database and that gives us what is called *weak consistency*: there are rules, but they are more like... guidelines?

Degree-Two Consistency. The idea of *degree-two consistency* is to prevent cascading rollbacks (aborts) without guaranteeing serializability. There are shared and exclusive locks, but two-phase behaviour is not required; in fact shared locks can be released at any time and locks can be acquired at any time, but exclusive locks must be held until the transaction commits or aborts [SKS11]. This means that we might read out of date data, but uncommitted values can never be read, so the level of transaction isolation here is "read-committed".

Cursor Stability. Cursor stability is a form of degree-two consistency for programs that iterate over some set of tuples using an iterator or cursor [SKS11]. This means that the tuples are examined or processed one at a time, in some order. To make sure that this works alright, the tuple currently being examined needs to be locked: before processing it is locked in shared mode; if any processing changes that tuple then it must first be locked in exclusive mode.

We don't get serializability of the transactions, but it can be a way to improve performance in a database where there are a number of relations that are popular and frequently accessed. But ultimately this offloads some of the work onto the applications using the database, meaning they must make sure they don't have a problem with non-serializable schedules [SKS11].

Users Ruin Everything... Remember the example earlier about selecting seats on a flight (or at a concert or sporting event...). When you go to select seats, you are presented with a map and you get to make your choice about which seat. You have some period of time, e.g., 2 to 10 minutes to get this done. That is a very long time as far as the database is concerned. We discussed the idea of using transaction isolation as a way of dealing with this problem: if two transactions select the same seats then we reject the second transaction.

This does have a major drawback when we're dealing with user-level timeframes. The database must remember information about updates performed by a transaction long after it has ended, for as long as any other concurrent transaction is still alive. In the case of transactions like a booking that give you 10 minutes of time, this means a transaction's information needs to be retained for up to 20 minutes.

A more likely solution is that we need to split it up into multiple transactions. One transaction is completed to read the data and it is provided to a client application (e.g., web browser). Then another transaction is needed when the user is ready to save changes; we create another transaction to save the data back to the database.

The data may have changed in the meantime, and if they did the attempt to merge the changes in that transaction will result in an error. What the application needs to do is reload the data (repeat the read(s) that produced the data elements and see if they changed. Detecting changes isn't as simple as just comparing fields, because it (1) would require saving the original read version, and (2) the ABA problem.

The ABA problem is not any sort of acronym nor a reference to the band ABBA. It's a value that is A, then changed to B, then changed back to A. The ABA problem is a big mess for the designer of lock-free Compare-And-Swap routines. It looks like nothing has changed, but actually it has.

The solution to the ABA problem is version numbers: every tuple is assigned a number that represents the version; it is initialized to 0 and incremented (atomically) each time the tuple is updated. Then a transaction needs only to look at the version number. If the version number doesn't match the original read, then it has changed. More formally, for each tuple modified in the current transaction, if the version numbers match then it may proceed and the version number is increased by 1; if there is a mismatch the transaction is rolled back (restarted, most likely) [SKS11].

This is called *optimistic concurrency control without read validation*. We assume we are going to succeed without conflict (are optimistic) and rollback if we must. It is without read validation, because we check version numbers only for writes that we are going to perform. If we want optimistic concurrency control, then we must also check the version numbers of any reads that went into the transactions too.

Concurrency in Index Structures

Rather than taking the simple approach and treating the index exactly the same as regular data elements, we can use special handling for the index structures to improve performance. More obviously, we can allow non-serializable access to the index as long as the results are consistent.

The first technique for locking on index structures is called the *crabbing protocol* [SKS11]:

When searching for a key value, first lock the root note in shared mode, and when traversing down the tree, acquire a shared lock on the child node and release the lock on the parent node, repeating this until a leaf node is reached.

When performing an insertion or deletion of a leaf node:

- Follow the same protocol to reach the leaf node (acquire and release shared locks).
- Lock the leaf node in exclusive mode and perform the insertion or deletion.
- If a split or coalescence is needed, lock the parent in exclusive mode and do the changes; release the leaf nodes.
- If the parent itself needs splitting or coalescing, retain the lock on the parent and propagate the changes; otherwise release the parent.

There is the possibility of deadlock in this protocol, but we know how to deal with deadlocks: restart a transaction and try again.

The second technique we will talk about requires a modification of the B^+-trees called *B-Link* trees. These have the added restriction that says every node (really, all of them) have a pointer to its sibling to the right. This helps in case a search is taking place at a time when a node is being split and it can continue. The locking protocol is then different for all operations as described below [SKS11]:

Lookup: Each node must be locked in shared mode before it is requested. A lock on any non-leaf node is released before a lock on any other node is requested. If a split takes place while a lookup is happening, then we might need to look in the sibling to find something (which is why there are sibling pointers). Leaf nodes are locked using the two phase protocol.

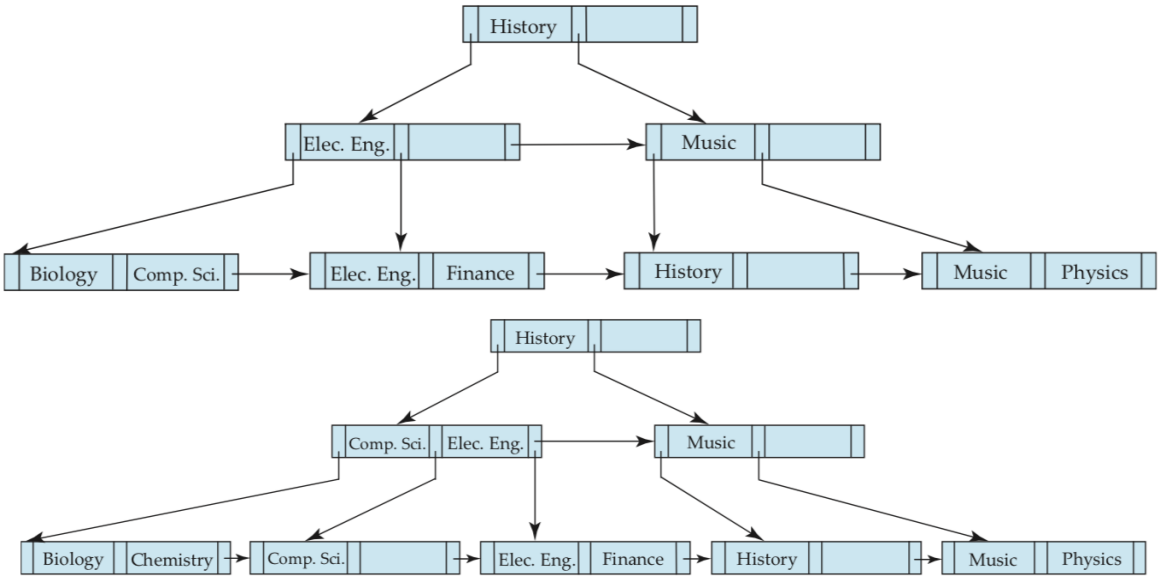
Insertion/Deletion: The rules for lookup are followed to find the leaf node where the records will be inserted or deleted. The lock for the leaf is upgraded to exclusive, and the insertion or deletion is performed. Two phase locking is used to avoid the phantom phenomenon.

If as a result of insertion or deletion, a split or coalescence is needed, the steps described below are followed to make that happen.

Split: If a node is split, a new node is created as per normal for a B-Tree and it becomes the right sibling of the original node (and the previous right sibling of the original is now the right sibling of the new node). Then the original node is released and an exclusive lock on the parent is requested to insert a pointer to that new node.

Coalescence: If a node need to be coalesced, then the node it will be merged with must be locked in exclusive mode. Then the parent node of the newly-created node must be locked exclusively so that the deleted node can be removed. The parent may be coalesced as well if necessary, otherwise it is released.

It's worth noting that it's possible that an insertion locks a node, unlocks it, and re-locks it again. Below is an example (from [SKS11]) of insertion of "Chemistry" into a B-Link tree.



Insertion of "Chemistry" into a B-Link tree (before above, after below). [SKS11]

So, running through the example: the insertion operation runs and it finds the destination where "Chemistry" should be inserted, which results in splitting the node at the bottom left and the creation of a new one.

Now let's see what happens if there is a lookup while this is going on at this point. The lookup gets blocked at the bottom-left leaf node because it is exclusively locked by the insertion. The lookup holds no locks at this point.

The insertion operations complete and relocks the parent node in exclusive mode, completing the insertion. This completely screws up the lookup operation because it is now holding a pointer to the wrong node. That's okay, because the lookup can start moving to the right through the sibling nodes until it finds the correct one. That seems a little bit painful, but it does mean it will succeed.

If instead of a split there was a coalescence, a lookup might be left holding the pointer to a node that no longer exists. That's okay, this lookup just needs to restart from the beginning. It's not ideal, but it does happen. To reduce the amount of times this happens, we might try to leave some nodes uncoalesced even when that would be the normal procedure; this breaks the rules about B-Trees, but it might be very brief if there are many more insertions than deletions.

Key-Value Locking. Key-value locking allows locking individual key values, allowing other operations to take place on values within the same leaf node and potentially increasing concurrency. The problem is that we could still have the phantom phenomenon, so the strategy of *next key locking* is employed: not only lock the keys that are in the range/search result but also whichever one is next [SKS11]. This prevent something from being inserted, altered, or deleted inside the search range.

28 — Data Warehousing & Mining

Data Warehousing & Mining

Databases, as you know, truly run the world. Think about a company like Amazon (let's restrict ourselves for the moment to their online store business). They have accumulated a huge amount of data about the purchases made. There needs to be a way to store this very large volume of data. That is what data *warehousing* is about. The purchase history of Amazon users between the years 2010 and 2015 is perhaps not needed on a regular basis, but it is saved and stored. But why? Because it can be mined: it provides insights.

Data mining is about extracting information from a large volume of stored data. The more data we have, the better predictions we can make. Amazon may know that 60% of the time, someone who buys product x will soon buy product y – and then can suggest y to you on the page when you have added x to your cart.

Now you might think that data mining is just helpful: Amazon is guessing about products I want and if they guess right, they are helping me! Maybe? But the truth is, this sort of data mining can be downright... creepy. Consider this story by Charles Duhigg, in the New York Times about how Target (the retailer) used the data it has to make predictions about whether a woman is expecting a child soon:

> Target has a baby-shower registry, and Pole started there, observing how shopping habits changed as a woman approached her due date, which women on the registry had willingly disclosed. He ran test after test, analyzing the data, and before long some useful patterns emerged. Lotions, for example. Lots of people buy lotion, but one of Pole's colleagues noticed that women on the baby registry were buying larger quantities of unscented lotion around the beginning of their second trimester. Another analyst noted that sometime in the first 20 weeks, pregnant women loaded up on supplements like calcium, magnesium and zinc. Many shoppers purchase soap and cotton balls, but when someone suddenly starts buying lots of scent-free soap and extra-big bags of cotton balls, in addition to hand sanitizers and washcloths, it signals they could be getting close to their delivery date.
>
> As Pole's computers crawled through the data, he was able to identify about 25 products that, when analyzed together, allowed him to assign each shopper a "pregnancy prediction" score. More important, he could also estimate her due date to within a small window, so Target could send coupons timed to very specific stages of her pregnancy.
>
> ...
>
> About a year after Pole created his pregnancy-prediction model, a man walked into a Target outside Minneapolis and demanded to see the manager. He was clutching coupons that had been sent to his daughter, and he was angry, according to an employee who participated in the conversation.
>
> "My daughter got this in the mail!" he said. "She's still in high school, and you're sending her coupons for baby clothes and cribs? Are you trying to encourage her to get pregnant?"
>
> The manager didn't have any idea what the man was talking about. He looked at the mailer. Sure enough, it was addressed to the man's daughter and contained advertisements for maternity clothing, nursery furniture and pictures of smiling infants. The manager apologized and then called a few days later to apologize again.
>
> On the phone, though, the father was somewhat abashed. "I had a talk with my daughter," he said. "It turns out there's been some activities in my house I haven't been completely aware of. She's due in August. I owe you an apology."

–Charles Duhigg [Duh12]

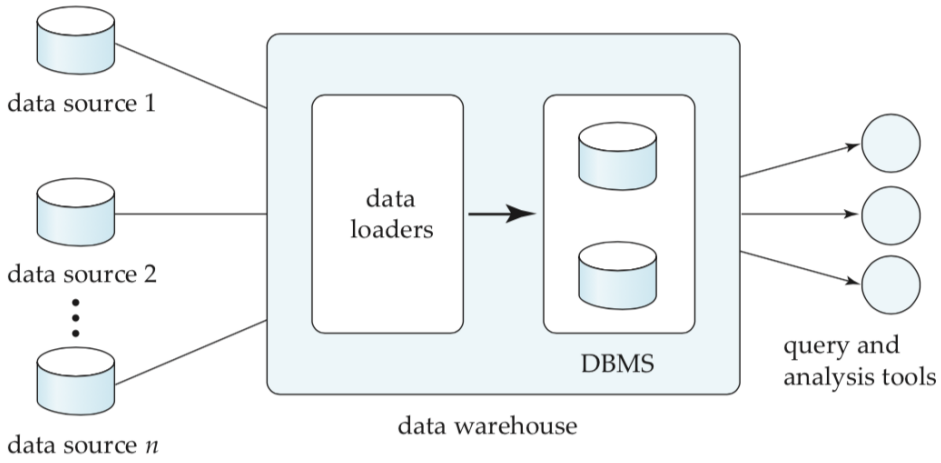
More generally, data warehousing is used to prepare large amounts of data for mining, and the purpose of the mining is to be used in what is called a *decision support system*. These are used not only to guess super personal information about shoppers, but also to tell store managers what products they should have in stock in what month, tell factory managers what products they should manufacture, and even (dare I say it) tell admissions officers what applicants should be admitted to a university [SKS11].

We'll first focus on how the data is to be aggregated (warehousing) before we get into a bit more about mining. The Target story makes for a compelling hook, but we have to eat our vegetables before we may have dessert.

Warehousing

It is virtually a certainty that in any large organization, the data will be spread across different databases or other sources. More than that, data can come from external sources: mailing lists from marketing organizations, credit scores from ratings agencies, legacy systems from purchased subsidiary companies, et cetera [SKS11]. To make things work, we will need to aggregate all this data somehow. The destination is the data warehouse, which contains large amounts of data in a common schema. There, the data resides for the long term and can be available for mining.

Consider a quick diagram from [SKS11] that overviews data warehouse architecture:



Data warehousing architecture [SKS11]

As we can see there are n data sources that feed into the warehouse. Each data source is associated with a "loader"; some sort of transformation routine that converts incoming data into the appropriate format for the warehouse's DBMS. Then the data is available for use in query and analysis tools.

The decision about when and how to gather data is important but perhaps boring. Data can be transferred from the original source to the warehouse periodically (e.g., every night), on demand (when a refresh is requested, automatically or manually), based on a certain volume of data (e.g., every n transactions)... There are many many options. Whatever we do, data is likely to be at least slightly out of date: today's purchases, for example, might not be in warehouse yet because they are not yet completed and closed. That is fine; even though the data may be slightly out of date we can still make good decisions or predictions based on what we have.

The decision of what schema to use in the warehouse will obviously depend on what the query and analysis tools are for. Importing the data requires transformation and with good luck all that is needed is a straight 1-to-1 mapping where this field corresponds to that field and all is well. Where things become difficult is when data is not so clean and consistent.

There is therefore the process of *data cleansing*: cleaning up minor inconsistencies in the data by correcting them [SKS11]. As examples, names and addresses can be misspelled (especially if a name is uncommon or uses

a less favoured spelling...) and so the same human may be represented by the names "Megan" and "Meghan" due to some entry error in one of the databases. We've already examined, to some degree, the idea of how we could repair such inconsistencies by examining the various probabilities of correctness of each value: if the name is "Meghan" in twelve places and "Megan" in one, we think it is likely the spelling with the "h" is correct.

You have probably already seen the idea of data cleansing on the small scale: if you have multiple contacts with the same name in your phone address book, it may suggest that you merge these contacts (if they are in fact the same person). This is de-duplication, that is, duplicate elimination.

Interestingly, we might want to use column-oriented storage in the warehouse rather than the standard row-oriented storage. It is noted in [SKS11] that there might be two advantages: (1) if we are fetching only a few attributes, we get exactly what we need rather than having to throw away parts of rows we don't need; and (2) storing things of the same type makes compression possible. But the drawback is that accessing a single row means multiple I/O operations.

Mining

With the data stored, now we want to use it. As we have already seen, in addition to statistical data, the data warehouse allows predictions or discovery of "new" information. There are three kinds of information we could discover [EN11]:

- **Association Rules**: One type of event is likely to take place at the same time as another: e.g., people who purchase a DSLR camera are likely to purchase a set of rechargeable batteries to go with it.
- **Sequential Patterns**: One event is likely to lead to another in the future: e.g., a person who buys a printer is likely to purchase a new toner cartridge within the next year.
- **Classification Trees**: Individuals (people) can be classified based on patterns of their behaviour: e.g., a person who purchases more than $x in a given year is a "top customer" and should receive some preferential treatment.

Association. We'll start off by looking at association. The example from [SKS11] we will use has to do with credit scores. To sum up how this works, a credit score is a way of assessing the likelihood that a person borrowing money will repay the money. Credit scores are represented by three digit scores generally, but they are obviously more complex than that. The best way of predicting whether or not someone is likely to repay a loan is, of course, their previous behaviour. But for someone who doesn't have a credit history (e.g., a person who has just graduated university) we might try to use some association data to predict (1) whether we should lend this person money and (2) if so, at what interest rate.

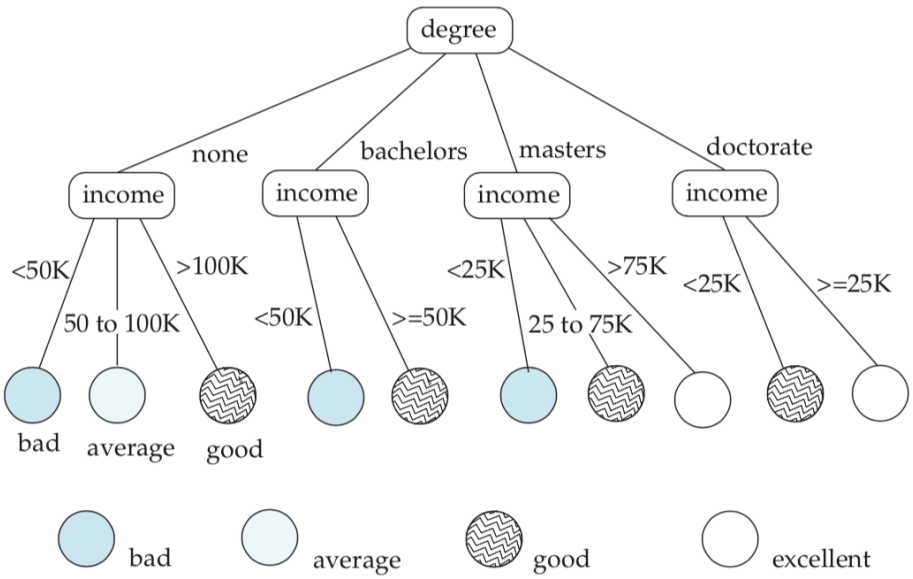
Suppose then there are four categories for credit: bad, average, good, and excellent. Then we need to figure out what are the key elements that determine someone's repayment history. We have some amount of data on an individual: place of birth, educational attainment, age, gender, income, marital status, and many more items. Some of the information we have is not relevant; some of it is. What we are looking for is a statistical correlation that says, for example, repayment of a loan is correlated with higher income. I'll assume you have a pretty good idea of how a statistical correlation is generated.

The data that we have is our *training set*. The training set is used to produce the rules by finding the correlations. What we should actually do is use a subset of the data as the training set. Then we verify the predictive power of the rules we have developed by looking at the rest of the data to see if the predictions match reality.

Now, in theory we could use every piece of available data to make a prediction, but realistically we only care about those elements that have a correlation and predictive value. And to prevent things from getting out of hand, we might pick only a top few items, ranking them from strongest correlation to weakest. Using those we can then build a tree that will be used to classify the creditworthiness of people for whom we have no (or limited) data.

It's worth noting that sometimes data may point to certain correlations, but it would be unethical to use it. Discrimination based on elements like family origin, gender, religion, anything like that is unethical if not illegal in a given jurisdiction. We might have the data in our database, though, or be able to find it out. It is still not okay to use.

In the creditworthiness example, let us assume the data we have says the biggest predictor of loan repayment is degree attainment, and the next biggest is income. If we are looking only at the top two elements (or all others produce a correlation below a certain cutoff) then we could construct a tree that looks something like:



A classification tree for credit risks based on degree attainment and income [SKS11].

It is important to note that these are predictions, guesses, really, and just because someone is predicted to have a good chance of repaying any loans made to them, does not actually mean that they will. We might say that good is a 75% chance of on-time repayment, for example, meaning that for 25% of people who would be predicted to be in the "good" category would end up paying late at least once or defaulting (missing payments).

Building Decision Trees. Our plan is to choose a sequence of partitioning attributes and use those to partition the data that we have so that we get "pure" leaves that contain only instances of a particular class. But there are good and bad ways to divide up the data. Suppose an insurance company wants to make some observations about who is at risk for car crashes. Criteria under consideration might be the colour of the vehicle insured and the number of horsepower of the engine. At first glance your intuition might say that it is obvious that the colour of the car is not very important. What we would like, however, is a way to quantify it: find out if it really has no impact or if it is significant. We can do so by treating this as an optimization problem: measure the purity of a data at the children nodes resulting by partitioning by that attribute. Choose the attribute that results in the highest purity.

We will now formalize this according to the definitions in the textbook [SKS11]. Suppose we have a set S of training instances and there are k classes; the fraction of instances in a class i is p_i. Then we can calculate the Gini measure as:

$$\text{Gini}(S) = 1 - \sum_{i-1}^{k} p_i^2$$

If all instances are in a single class, the Gini measure is 0; its maximum value is $1 - 1/k$ if every class is equal in size. This can be used to calculate the information gain; how much we have learned. If S is split into multiple sets, the purity of the split is calculated as:

$$\text{Purity}(S_1, S_2, ...S_n) = \sum_{i=1}^{n} \frac{|S_i|}{|S|} \text{purity}(S_i)$$

Then information gain is calculated as purity(S) $-$ purity($S_1, S_2, ...S_n$).

Another measure is of entropy which is:

$$\text{Entropy}(S) = -\sum_{i-1}^{k} p_i \log_2 p_i$$

The entropy value is 0 if all instances are in a single class; and has a maximum value if all classes are equal in size. From that we can calculate information content as:

$$-\sum_{i-1}^{n} \frac{|S_i|}{|S|} \log_2 \frac{|S_i|}{|S|}$$

And the best split for an attribute is then the one that gives the most benefit: information gain divided by information content.

Bayesian Classifiers. Our choice of correlation criteria does not necessarily account for correlations between the criteria themselves. For example, degree attainment is correlated with higher income in general and there may even be a causation relationship there. It doesn't really matter (except to statisticians); all that we are concerned about is the correlation, the predictive power. Whether higher income causes a person to be more able to repay a loan, or a person who is more likely to repay a loan is more likely to earn more money, or some other third factor increases both is not relevant for our purposes.[21]

To put a little more formality to this discussion, we will use Bayes' theorem: $P(A|B) = \frac{P(B|A)P(A)}{P(B)}$. Here's a quick example from [SKS11] where only income is used as a predictor. Suppose that a person's income is $76 000. This means they fall into the bucket for the range $75 000 – $76 000. Looking at people who have a rating of excellent, the probability of income being in that range is 0.1; looking at people who have a rating of good, the probability of income being in that range is 0.05. And if the overall fraction of people in excellent is 0.1 and the overall fraction in good is 0.3 we can plug and chug: the probability of excellent for this person is .01 and for good it is 0.015. The highest probability is 0.015 so we go with that: this person is predicted to be in the category good, and that is the category they are assigned.

We mentioned it informally, but formally we need to address the fact that attributes may be related. The conditional probability of any particular attribute is highly dependant on the other attributes. In the worst case scenario, we would need to draw from a pool of people who are exactly alike on all relevant attributes (e.g., age, gender, postal code, income, degree level) to ensure that the prediction is good. If there are very few people who match in all ways (or no people) then the prediction is likely to be garbage. So we will assume that all attributes have independent distributions: i.e., we can simply multiply them together. We know that isn't true, but it is sufficient for the job of estimation. It just might make a stats prof cry.

There are multiple algorithms for figuring out how to classify elements. Textbooks may discuss several choices for how to get this done, aside from the statistical method we have discussed.

Association. Remember that association is about events that are likely to take place at the same time; e.g., a customer who buys one item is likely to buy another. We are basically, once again, looking at statistics here. If the user adds the Blu-Ray disk for "Inception" to their cart, then to present a list of suggestions, we look at the history of purchases: what are the three items that appear in shopping carts most often alongside that film?

Rules for association have an associated *support* as well as *confidence*. Let's look at a quick shopping example from [SKS11]. We have a simple corner store and it sells some basic supplies.

Support is what fraction of the purchases satisfy the antecedent and the consequent. Remember that if the statement is P implies Q, the antecedent is P and the consequent is Q. So if the premise is that buying milk implies buying bread, milk is the antecedent and bread is the consequent. If it so happens that 0.001% of all purchases

[21] Obligatory xkcd: `https://xkcd.com/552/`

include both milk and screwdrivers, then support for the proposition that milk implies screwdrivers is very low. If, however, 50% of all purchases include both milk and bread, support for this rule is very high.

Confidence is how often the consequent is true when the antecedent is true. The proposition of bread implies milk has a confidence of 80% if 80% of purchases of bread also include milk. A proposition with low confidence is probably not useful. Keep in mind that P implies Q can have a different confident from Q implies P, even though they have the same support. That is to say, it might be the case that 100% of purchases for battery chargers include batteries, but only 10% of purchases of batteries include battery chargers.

In short, we want to find the items that are most likely to be purchased together. So first we find the sets of items that have large support (e.g., are likely to be purchased together). Then we look at the confidence level inside each set. Only those with sufficient confidence are offered up as suggestions.

Data Visualization

To wrap up, a small digression about data visualization. A visualization system helps users to look at large volumes of data and detect patterns. If data is presented well, then humans can see patterns in the data that might not otherwise be easy to detect. Bar charts, heat maps, and line graphs can all be effective ways of representing the data.

What is important, though, in data visualization is: (1) choosing the right data to show and (2) showing it in a way that is easily comprehensible.

29 — Recovery

Recovery

It's not a question of **if** things will go wrong, but a question of **when**. Bad things will happen and it matters how we deal with them. There are stories of tech companies who have a process specifically designed to cause random things to fail and crash so that the system is always designed to deal with failure at any time. That's extreme, but we need to be prepared. There will be failures: power failures, hardware failures, user failures...

In [SKS11] there are a few different types of failure that are worthy of our consideration:

- **Transaction Failure: Logical Error.** A transaction cannot continue because of an internal condition: invalid value, overflow, resource limit exceeded...
- **Transaction Failure: System Error.** Something has gone wrong and the system is in an undesirable state and the transaction cannot continue but can be restarted. A typical example here is deadlock.
- **System Crash.** A hardware malfunction, bug in the database or OS, that causes the loss of volatile storage contents and all transactions are halted; nonvolatile storage is not affected.
- **Disk Failure.** Allegedly-stable storage suffers a failure; copies from backups are needed to restore the data.

As is typical in a lot of programs, an error or bug brings the system down immediately, which is called "fail-stop" behaviour. This is normal behaviour in a C program, for example, but it may be even more important in the database. The fail-stop assumption is important because it prevents (or minimizes) corruption of the data. If we let the system carry on in an inconsistent state, we risk propagating invalid or corrupt data and infecting, so to speak, the rest of the data. Fail-stop prevents this risk, even if it is sometimes frustrating.

If something goes wrong, we want to "recover" from that failure. Recovery is this sense is, as you will recall, putting the database back into a consistent state. For that to work we need either to have some sort of log, or some sort of backup [EN11].

If things are really bad the recovery method is to restore from backups. We copy the data from the backup storage media and then restart the database server with that restored data set. Any data that has changed between the time of the backup and the time of the crash is quite simply lost. So it is important for backups to be current...

Supposing things are not so bad then we can use the log to try to recover; whether particular operations are undone or re-done to restore the consistent database. Looking at the log tells us what transactions were in progress, and based on what information is available, we can choose to carry out the changes or roll them back. For a transaction that was not complete, roll it back. For one that committed but the data changes have not yet completely been copied out to disk, the changes are carried out.

If we do *deferred updates* then the data on disk is not updated until after the commit point of the transaction. If that is the case, then the changes are noted down in the database log, and the log is written out to disk, and then and only then can we proceed. This strategy is, if you recall, mostly the same as how things work in the Windows NTFS file system. If a transaction has failed in some way before the commit, no changes in persistent storage have taken place. It may be necessary to redo steps that are stored in the log if the failure takes place after the commit. This is sometimes called the "no undo / redo" algorithm [EN11].

The opposite approach is *immediate updates*; the data is updated before the commit but still the operations are recorded in persistent storage before they take place. If the failure occurs before we get to the commit point, then we have to roll back and undo some changes of the transaction. It turns out that under this algorithm both undo and redo operations may be necessary so it is called the "undo / redo" algorithm. A variation that requires us to make all database changes happen in persistent storage means that we only have to undo things, leading to the "undo / no redo" algorithm [EN11].

It is also noted in [EN11] that we should remember that recovery can fail too. That means the undo and redo operations need to be *idempotent*: executing the operation multiple times has the same effect as executing it just once. If we need to restart the recovery process, we don't want inconsistent results. An operation like `x = 7;` is idempotent because that operation can be repeated arbitrarily many times and the value will remain 7. On the contrary, `x++` is not idempotent because every execution changes the value of `x`. This is why, for example, we would want to record the operation in the database log as being something like an assignment statement, or even better, something that reads like "x is changed from 6 to 7".

Writing to Disk.

If we really want to be sure that a write has succeeded, we need to check; as Ronald Reagan said: "trust, but verify". A transfer may succeed completely, have a partial failure (some data in the block is changed but not all), or suffer a total failure (the destination is not changed at all) [SKS11]. If something goes wrong we need to actually detect it, of course, to know that something has gone wrong.

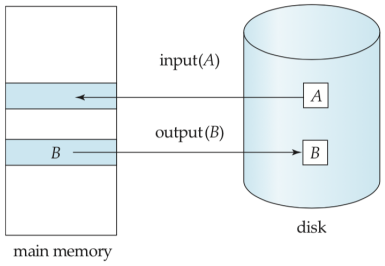
One way to be sure is to have two physical blocks for each of the database logical blocks (and even better if they are not on the same media). The process for a write is really to write the first block, check that it succeeded, write a copy to a second block. After the second write has happened, the write is finally considered completed.

If a failure takes place during the writing of either one of the blocks, they may not match. If they are the same then no error is detected and we don't have to do anything. If there is a mismatch then we can look at the checksum of each disk block (you do have those, right...?): the block with the invalid checksum is the one that is wrong and the other one will be treated as correct; the correct overwrites the incorrect. If both checksums are valid, the second block is copied over the first: if the write did not complete the full process and update both blocks, we consider it failed and roll it back [SKS11].

Caching (Buffering) of Disk Blocks

As we have already examined, there are often disk blocks in memory being cached, and the recovery process's success or failure may depend on how the data in the cache is handled. Because the cache data is in volatile memory, a crash or loss of power causes it to wink out of existence. Normally, management of paging memory out to disk and back into RAM is the purview of the operating system, and probably you learned about that extensively in an examination of the same. A typical program doesn't need to manage this aspect of memory, but a database server does.

Conveniently, operating systems make some low-level system calls available to allow this to happen. We'll call those operations on a block B `input(B)` for reading a block into memory, and `output(B)` for writing a block to disk (replacing what was on disk already in that spot).



Block storage operations. [SKS11]

In addition to this, the database has a small directory of what items are in the buffers (which resembles a page

table in operating systems, actually) [EN11]. If some item is needed for a read or write, the directory is checked to find out if what we need is already in cache; if it is not then it must be retrieved from disk to be used. It might also be necessary to flush some of the data out to disk if there is no more space and we must replace a block.

That sort of "normal" writing out to disk has some restrictions. Previous discussions of caching have covered the various algorithms and keeping track of whether the block was modified, but there exists an additional restriction. This is whether or not the block is *pinned* – that is, forced to remain in memory and not permitted to be written to disk. This is often implemented as just a single bit in the block.

A very common example of why a page might be pinned by the recovery system is because it has been updated by a transaction that has not been committed [EN11]. If we wait for the transaction to commit, the block in question won't be written out to disk in a partially-updated state.

Writing a block to the output via the `output` command itself might not result in immediately writing a particular block to disk, because that block still might be needed for other operations. That can cause a bit of a headache because a crash before the output operation takes places means the data is lost... The recovery system can choose, however, to insist a block be written to disk immediately, and that is called a *force output* if a write command is explicitly issued [SKS11].

Ensuring Atomicity

What we really want to avoid is that a transaction is incompletely recorded in the non-volatile storage, violating the atomicity of the transaction. The example we'll use (from [SKS11]) is the standard banking sort of example: we want to transfer \$50 from account A (initial balance, \$1000) to account B (initial balance, \$2000). If the system crashed during the execution, after the write of the block containing A has taken place but before the write of the block containing B has, things get interesting. Volatile memory is lost in the crash so we don't know the fate of the transaction.

Under these circumstances, looking at the data usually does not tell us enough. If A shows a balance of \$950 and B a balance of \$2000, we know (from the setup of the example) that this is incorrect. The database system, however can't tell if those values are correct or incorrect without some additional information (just as we can't either). And as we have already discussed, we can solve this by writing the information to the log before performing the actual write.

More formally, the log is a tuple of 4 elements: T_i, the transaction in question; X_j, the data element to be modified; V_1 the old value; and V_2 the new value. Other log records would indicate when a transaction begins, when it commits, or when it aborts.

To prevent the log size from getting out of hand, we will need to eventually delete some old history. We could take the NTFS-like approach and delete transactions from the log once they are well and truly finished, but that, as we will soon see, is not actually what we do.

These give us enough information to perform the undo or redo operations as described above. In the case of the bank transfer example, we can see that the step to change the balance of A from \$1000 to \$950 has succeeded but the step to change the balance of B from \$2000 to \$2050 has not. There are two valid paths for proceeding: undo the change to A OR redo the change to B. Both produce consistent results. You might imagine that it is always preferable to redo the operation, but that can really only happen if the transaction has already committed... otherwise we don't know what else the transaction was going to do.

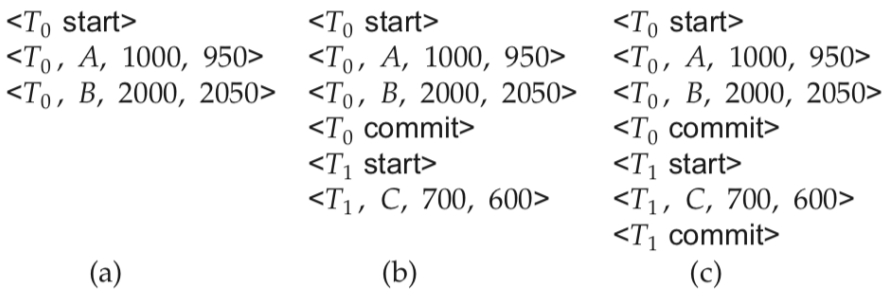
As you might imagine, if there are multiple things to be undone or redone, the order matters to produce the correct result. If you don't believe that order matters, next time you bake a cake, first bake all the ingredients, then mix them. The redo approach then, requires us to scan the log from top to bottom, performing appropriate redo actions where necessary on each log record [SKS11]. This is efficient and sensible, preserving order and reading the log only once, sequentially.

The undo operation works a little differently: it restores data items to their old values, but also writes a log noting what it has done. The log records created are noted as "redo-only" because they don't contain the old (now-erased) value of the log record [SKS11]. That looks like it's very strange, but we'll see shortly why this is so. And when all undo operations for a given transaction are completed, an abort log record is put into the log to show that the

undo is finished. This guarantees that every transaction will have either a commit or abort record in the log, even those that were unfinished at the time of the crash.

With these rules in mind, given a log, how do we determine what transactions need to be undone or redone? Obviously, if a transaction does not have either a commit or abort log message it was unfinished and will be aborted. If a transaction has a commit in the log, it needs to be redone. But if a transaction has an abort in the log it needs to be redone as well. Wait, what? When a transaction is undone, as above, we need to undo the changes with those redo-only log entries to make sure everything is put back where it should be.

Let's look at an example:



A transaction log at three different points [SKS11].

If the system crashes at point (a) in the above diagram, transaction T_0 is started but has neither committed nor aborted. We need to therefore, undo the operations it has done. So the following log entries will be created: a log entry setting B back to 2000, an entry setting A back to 1000, and a record noting that T_0 is aborted.

If the crash takes place at point (b), then transaction T_0 is complete but T_1 is not. Right away we can conclude that we need to abort T_1 and roll it back as above. So C is set back to 700. As for T_0, we will need to redo the transaction and we will see A at 950 and B at 2050 in the end.

Should the crash take place at point (c) in the diagram, then both T_0 and T_1 are committed and need to be redone. The final values: A is 950, B is 2050, C is 600.

Remember that if a transaction aborts, the transaction is "redone" in the sense that redo-only statements are carried out to cancel out anything that was done before it. So the transaction might write a value to X only to have it reversed by the subsequent operation before the abort.

Those who do not forget history are doomed to repeat it. [22] As just mentioned, the log structure we have means we write a lot into the log but never take anything out. And in theory to recover from a crash we have to scan the entire log and redo or undo anything needing to be redone or undone. This can be, however, a very long list. Repeating the redo operations many times does no harm, but it is a huge waste of time.

What we'd like is to remove from the log any transactions that are well and truly done and we're sure a redo operation is totally redundant. We can remove them one at a time as the very last step of actually recording a transaction as done, but that can require a lot of searching the log to find which one is the right one to remove. Instead, we can try to do this en masse through *checkpoints*.

A checkpoint is a particular place in the log where we "save our progress", so to speak, so that if there is a crash we need only resume from the last checkpoint rather than starting at the beginning of the log. You are very likely already familiar with checkpoints, because they have been something you encountered in another context. Suppose everyone's favourite Italian plumber plunges to an untimely death: you can resume from some partial progress, such as the beginning of the level, or even some halfway point within the level, rather than having to start the quest to save the princess from the very beginning.

The steps to perform a checkpoint are [SKS11]:

[22] With apologies to George Santayana.

1. Write all logs currently in volatile storage to stable storage.
2. Write all all modified buffer block to stable storage.
3. Place a checkpoint record in the log.

Transactions are not allowed to make changes to buffers or logs during the time a checkpoint is being created (although we might be able to relax this rule).

The checkpoint record notes that it is a checkpoint and has a list L of all transactions that are active at the time of the checkpoint. The checkpoint means that any transaction before that point in the log is completed and for sure put out to stable storage, so those transactions do not need to be repeated. Excellent.

If a crash occurs, we just look for the last checkpoint in the log (the one closest to the end of the file) and then the only redo and undo operations that need to be performed are those on the transactions in the list L and any that started after the checkpoint. From there the routine works as we have previously described: for each transaction, redo or undo depending on the state of the transaction.

The checkpoint also allows us to get rid of some records once they become sufficiently old: anything that is before the first transaction in L of a checkpoint is "ancient history" and can be removed from the log.

The Recovery Algorithm

The hall is rented and the orchestra engaged: now it's time to see if we can dance[23] Now it is time to actually execute the algorithm!

Recovery takes place in two phases, first the redo phase and then the undo phase (really). We start at the last checkpoint in the log and scan our way forward, then reverse direction and go back. The algorithm is described briefly in [SKS11]:

The *redo phase* requires replaying updates of all transactions, scanning forward in the log from the last checkpoint. Even the ones that were rolled back (as we have seen) and including those transactions that were incomplete. We still do those operations, but we will note for later the incomplete transactions.

More formally:

1. The list of transactions to be rolled back is set to the list L of the last checkpoint record in the log.
2. Each log operation is then carried out, both the "regular" and redo-only operations.
3. If we encounter the start of a transaction marker for a new transaction, add that transaction to the list of transactions to be rolled back.
4. If we reach an abort or commit statement for a transaction, remove that transaction from the list of transactions to be rolled back.

Following this phase of the process, we have identified all transactions that were incomplete (neither committed nor aborted) at the time of the crash.

Then we have the *undo phase*: roll back all appropriate transactions in the list we have just generated in the redo phase. We scan the log backwards from the end moving up.

1. Whenever we find a record in the undo list, we perform the usual undo actions as if it was rollback of a failed transaction.
2. When we reach the start log entry of a transaction we know we have undone all actions of that transaction, so we can put an abort record at the end of the log, and remove this transaction from the list.

[23] Watch the episode of Star Trek: The Next Generation called "Q Who".

3. The undo phase terminates when the undo-list is empty (i.e., all incomplete transactions are properly undone).

Following the undo phase, the database is in a consistent state again and new transactions can be permitted to begin... in other words, the system may begin normal operations. The database is once again in a consistent state and all transactions have the atomic nature they promise: they have either completed successfully or it is as if they haven't happened at all. Some operations were cancelled and potentially permanently lost, but the database is in a consistent state and that is the goal of the recovery process.

30 — More Recovery & ARIES

Fuzzy Checkpointing

Thus far when we have talked about creating a checkpoint assumes that we stop all execution, briefly, to take the snapshot. This sort of thing is not unheard of in programs, such as the "stop the world" garbage collector behaviour in Java. It might be, however, undesirable or even unacceptable: for a sufficiently large buffer it might take a very long time. Therefore we would like to modify the checkpoint system to avoid having to halt all execution.

The term for a checkpoint that isn't such a clean division between here and there is called a *Fuzzy Checkpoint*. A fuzzy checkpoint allows updates to start once the checkpoint has been created but before the modified buffer blocks are written to disk [EN11]. There's still a pause in execution while the checkpoint is created, but that should be the fast part.

This does potentially present a problem: the checkpoint data is written to disk before the blocks are written and we could have a crash before all the blocks are written to disk. What do we do then? Keep track of the last completed checkpoint, and don't update the last completed checkpoint marker until all blocks have been output.

Note that even with fuzzy checkpoints, a block cannot be updated while it is being output to disk (although other blocks can be modified) [SKS11]. Fuzzy checkpoints are more or less the standard approach in the real world because the alternatives take too long.

Recovering from Loss of Nonvolatile Storage

The recovery routines considered so far are about booting the server up again after a crash or power failure or other temporary outage where nonvolatile storage is unaffected. But bad things will happen to even nonvolatile storage. So we need to take copies of the data to account for the fact that even supposedly nonvolatile storage is lost.

But how do we back up the database data? It is called a *dump*. It is basically taking a copy of all the data. If we are doing it live, we require that no transactions are running and the steps are [SKS11]:

1. Write all log records currently in main memory to stable storage
2. Write all buffer blocks to disk
3. Copy the contents of the database to stable storage
4. Note completion of the dump in the log

The form the dump takes could be raw data, but there are tools in most database packages that exports the data as a SQL dump: it writes out the data description language (create table statements, for example) and insert table statements for the data. Such a dump can be used to copy the database from one system to another or migrate it from one database server to another.

The approach described above does have the obvious downside of needing to stop execution for a very long time. That is, however, not realistic. A tool that dumps the database like `mysqldump` takes a complete database dump,

but it does so as a transaction: it gets a snapshot of the data at the time the dump is requested; further changes can still go on in the meantime but this transaction will be left running for as long as it takes (and it can be HOURS!).

An import will need to be run to get the data back into the database from the dump file if it was in SQL format. In the case of a loss of all data then it is imported into a fresh (empty) database; otherwise the previous (corrupt, partial, or otherwise problematic) data is dropped and it is replaced with the data from the import.

Exporting seems fairly easy from a technical perspective: just copy the data to the output file in the form of create table statements and then insert into statements. That works but there is a problem when importing. Tables are added and they have foreign key constraints. But how can you set up the key constraints if the other tables don't exist? And how can you insert the data for one table that references another without that other table having the data present as well?

There is some possibility to do a topological sort where you find a sequence of create table and add constraint statements, then one for insert statements... This is very difficult and potentially impossible depending on how your data is defined. The real answer is that when importing data via this method, integrity constraints are temporarily disabled to allow the file to be processed sequentially without any slowdowns.

Another way that backups tend to get taken is at the level outside of the database: the operating system. The data is ultimately stored somewhere on a volume (or volumes) and the operating system can do things like shadow copies and image backups of the drives. Of course, the state that is stored to the disk at the time of the backup is what gets copied. If this is not a consistent state, some additional actions may be needed on startup to get things back to a consistent state.

ARIES

In spite of sounding like the name of a NASA program to Mars or some such, the ARIES algorithm is a database recovery routine used by IBM. It relies on three main pillars: write-ahead logging, repeating history during redo, and logging changes when undoing something. The description of how ARIES works is all from [EN11].

The algorithm and the approach we have examined for recovery works and it produces the correct results, but it comes with some drawbacks and performance decreasing elements. ARIES is hopefully going to address it. ARIES is broken up into three phases: analysis, redo, and undo.

Analysis. In the analysis phase, the goal is to figure out what pages in the buffer were dirty. We don't have the actual pages, of course, because the crash caused the loss of that data. However, we can determine the information based on the log data from the checkpoints: the dirty page table, transaction table, et cetera. During this phase a determination will also be made as to where to begin the redo phase.

Redo. In the redo phase, updates from the log are done in the database. The normal recovery routine expects that we only redo committed transactions, but in ARIES we will try hard to skip unnecessary work: only redo operations that are necessary.

Undo. The last phase is the undo phase, and the log is scanned backwards and transactions that were active at the time of the crash are undone in reverse order.

Every log has a LSN – *Log Sequence Number*; this is an incrementing counter that indicates the log record's address on disk; each number corresponds to a specific change of some transaction. Each data page stores the LSN of the most recent log record that changed that page. Log records are generated for updating a page, committing a transaction, aborting a transaction, undoing an update, and ending a transaction. When an update is undone, a compensation log record is added; when a transaction ends, successfully or unsuccessfully, an end log record is written.

The log structure has numerous fields, including the previous LSN for that transaction, the transaction ID, and what type of transaction is being written. The LSN can link (in reverse order) the log records for the same transaction. Other fields in each log record might include where to find the item (page, offset, etc), as well as the before and after values.

To perform a recovery, we need the log as well as the *transaction table* and *dirty page table*, both of which are maintained by the transaction manager. The transaction table lists the various transactions in progress. Similarly, the dirty page table contains the pages that are dirty (shocking, I know). These are lost when a crash occurs but are rebuilt when the system comes back online, obviously by looking at the log data. Those tell us about what information is in progress.

To take a checkpoint, some steps are the same as the regular checkpoint process: a begin-checkpoint entry is added to the log, the checkpoint is written, and then the end-checkpoint entry is added. Alongside the end-checkpoint entry, the data for the transaction table and dirty page table are written to the log as well. The last step is to add the LSN of the begin-checkpoint entry to a special file. This special file just marks the location of the last successfully completed checkpoint operation and it will be checked during recovery so we know where our last checkpoint is. Note that fuzzy checkpointing is used so the cost of writing all this data out is not so high. If a crash occurs during the checkpoint process, the special file will not yet have been updated so we will resume from the earlier point.

After a crash occurs, the recovery process begins. The first thing is to use the special file just mentioned to find the place in the log where the process starts. This is where the analysis phase begins: start with the begin-checkpoint entry and keep moving until the end of the log (with stops along the way).

When the end-checkpoint entry is found, the transaction table and dirty page table are read into memory. Their content will be altered in memory as log processing continues: if a transaction T was in progress at the time of the end of the checkpoint and we then encounter an end-transaction record for T then we know that it completed (successfully or otherwise) and we can remove T from the memory list of transactions in progress. If instead we found a transaction T that was not in the in-progress transaction list at the time of the end checkpoint, we will add it to the transaction table.

Whether the transaction was known or not before encountering an operation involving it, anything other than an end-transaction statement will result in updating the LSN for that transaction in our records. If the transaction's operation results in a change to a page P, then P is added to the dirty page list (if not already present) and the LSN field for P is updated as well.

At the conclusion of the analysis phase, the transaction table and dirty page table have all the information necessary to make the changes to return the system to a consistent state.

The redo phase begins: to avoid having to do anything unnecessary, no changes are done that have already been saved to disk. It is easy enough to tell which changes are done: look at the dirty page table, and find the smallest LSN, which we will call M. Anything before that is already done, so we can skip over records in the log that have a LSN smaller than M (because they are already done or overwritten in memory). Redo starts then at the LSN equal to M and scans forward through the log.

For each change, consider whether the page P is in the dirty page table. If it is not, then the change is already on disk. If P is in the dirty page table, but has a LSN larger than the current change, the change is already written to disk and does not need to be reapplied. If P is in the dirty page table but does not have a larger LSN, page P is read from disk and the disk stored version is checked; if the LSN is larger than that of the operation being performed then we can skip this operation. If none of those conditions led to skipping the operation then the redo is applied.

That concludes the redo phase; at this point the database is in the exact state it was in at the time of the crash, but the state is not consistent. There are still uncompleted transactions with partial progress that need to be rolled back. That is the job of the undo phase.

The log is scanned backwards from the end and each operation by a transaction in the "undo set" is undone by entering a compensating transaction record. The undo set is all transactions that were in progress at the time of the last checkpoint, plus all those that started after the checkpoint, subtracting those that finished before the crash (regardless of whether they were in progress at the checkpoint).

Once that is done then the database is restored to a consistent state and normal operations may resume.

Let us now do a simple example of how ARIES works from [EN11]. Let us assume we have three transactions T_1, T_2, T_3. T_1 updates page C, T_2 updates B and C, and T_3 updates A. We then have a crash occur, and the state

of the system is as shown in the two diagrams below:

Lsn	Last_lsn	Tran_id	Type	Page_id	Other_information
1	0	T_1	update	C	...
2	0	T_2	update	B	...
3	1	T_1	commit		...
4	begin checkpoint				
5	end checkpoint				
6	0	T_3	update	A	...
7	2	T_2	update	C	...
8	7	T_2	commit		...

The log at the point of the crash [EN11].

TRANSACTION TABLE

Transaction_id	Last_lsn	Status
T_1	3	commit
T_2	2	in progress

DIRTY PAGE TABLE

Page_id	Lsn
C	1
B	2

Transaction Table and Dirty Page Table at the time of the crash [EN11].

With this information in hand we may begin to execute the algorithm. The first step is the analysis phase. The first step is to go to the most recently completed checkpoint. That so happens to be at LSN 4. So analysis moves from there forward.

LSN 5 is the end checkpoint statement so it contains the transaction and dirty page tables. In LSN 6, transaction T_3 is found and added to the the list of transactions in progress. LSN 7 doesn't tell us anything new, but LSN 8 tells us that the transaction T_2 has committed. So the analysis phase is complete and the result is shown below. It's worth noting that transactions T_1 and T_2 have status "commit" but still appear in there – because the pages they have updated have not been written to disk.

TRANSACTION TABLE

Transaction_id	Last_lsn	Status
T_1	3	commit
T_2	8	commit
T_3	6	in progress

DIRTY PAGE TABLE

Page_id	Lsn
C	1
B	2
A	6

Transaction Table and Dirty Page Table after analysis [EN11].

Then it is time for the redo phase: the smaller LSN in the dirty page tables is 1 (page C) so that's where we start the redo process. LSNs 1, 2, 6, and 7 want to update pages. For those pages that have higher LSNs than the updates, no need to repeat that update.

The undo phase here is very simple: the only thing to undo is the operations of T_3. We go backwards and the only log entry we need to undo is LSN 6. At that point we are all done.

Remote Backups

Somewhat related to the idea of recovering our database if we have a crash is the idea of an online backup: a second copy of the database standing by to take over if the primary one should go down (whether through a crash

or by losing network connection). The second copy is certainly on different hardware, and should more likely be located at a different physical location.

By moving it to a different physical location, of course, that adds some latency to the connection: it takes nontrivial time for data to be transferred from the primary to the backup location. And that means that a strategy for keeping these in sync is necessary, but there are other considerations:

Detecting failure. As you might imagine, the backup system needs to detect that the primary has failed so it knows when to take over. Ideally there are multiple independent communication lines between the primary and the backup so that the failure of any single one is not mis-detected as a failure of the primary system [SKS11].

Taking over. Suppose a real failure is detected and the primary has crashed. In the meantime, the backup does the job of the primary. If the first system comes back online, it could be the new backup or it could retake the role of primary. Either way, it needs to get caught up on the things that have happened in the meantime. Once both systems are back online, the redo logs can be sent to the system that is behind to get it caught up [SKS11].

Making extra sure. In this sort of configuration, is a transaction really committed until it has reached the backup site? It turns out we get a choice of how much we are willing to tolerate [SKS11]:

- **One-Safe**: If the transaction has been written to stable storage at the primary location. If there is a crash at the primary before that transaction has made it to the secondary location, then it may appear to be lost. When the primary site comes back, it may happen that in the meantime there has been an incompatible change made only on the secondary. Imagine you book seat 4A on the primary system and it crashes; someone else then books 4A on the secondary system. Who gets the seat? This is something that may require humans to sort out.
- **Two-Very-Safe**: A transaction is only committed once it is in both the primary and backup stable storage... which means that no transactions can be executed if one site is down.
- **Two-Safe**: A transaction is only committed once it is in both the primary and backup stable storage... if both are up; otherwise it is the same as one-safe: the primary suffices.

Either two-safe or two-very-safe necessarily comes with a performance penalty: when the transaction is committed the changes must be propagated to the other server (if up) to consider the transaction fully done.

We're not limited to only two-safe protocols, of course. We could insist on an n-safe protocol where we have some additional replica servers and we consider the transaction done only when all n servers have completed the transaction. The cost of having n servers contain the data is really not much higher than the cost of having two: we have to send the data out and wait for confirmation, but much of this can be done in parallel.

But if we are going to be sending data out to multiple databases... are we sure they agree? We will need to come back to this subject next...

31 — Recovery: Repairing Inconsistent Data

Inconsistent data? In my database? It's more likely than you think!

Inconsistent databases are a difficult problem for database designers to deal with. As the definition of inconsistent suggests, one or more integrity constraints have been violated, and one might conclude that queries on the database would be impossible or the answers meaningless. Research on the subject, specifically in the fields of repairing inconsistencies and answering queries despite the status of the database, has provided several options. The works we examine restrict themselves to relational databases, and not any of the other possible types.

We cannot proceed without first providing a working definition of what is meant when we say that a database is consistent. We say a database D is *consistent* if D satisfies the set of integrity constraints C in the standard sense [M. 99]. If any of these constraints are not met, the database is *inconsistent* (or has inconsistencies). When there are inconsistencies whereby there are multiple tuples reflecting the same real-world entity, we name that a *cluster* [AFM06].

The obvious source of how databases may become inconsistent is: errors. Data can be entered that violates integrity constraints. That can happen if the database was implemented without certain important integrity constraints (e.g., foreign keys) added, or a new rule is being added. It can also happen because database integrity constraints can be turned off (yikes!) which is usually intended for mass data import, or perhaps because the database engine (myISAM anyone?) just does not enforce those constraints. So in all cases, the inconsistency was added in error.

However, it is not the case, that by simply relying on error-detection and rule enforcement, we can be certain that no inconsistencies will ever arise. We will look at some ideas of how databases may become inconsistent in the first place, aside from the obvious answer of errors. Papers [M. 99] and [BC99] offer some plausible scenarios where inconsistencies may be introduced. The first is from data warehousing; when data is imported from various sources and may need cleaning before storing. The second is database integration; when some data is stored across many databases stitched together, there may be different constraints at a local level than at the higher level, resulting in a locally consistent, yet globally inconsistent, database. It is possible that integrity checking may simply be too costly from a performance standpoint, and the database must proceed without it. Finally, we might, under some circumstances, choose to allow a temporary inconsistency, such as warehouse stock levels being allowed to fall below a minimum when replenishments have already been ordered. It is clear from these three scenarios that there exist plausible reasons why a database might be inconsistent, and why this subject is not rendered irrelevant in an error-free system.

We would like the answer from a query against an inconsistent database to be the same as that of the consistent database, but with partial information, we may not know what the consistent database would return. At the very least, we are interested in getting a consistent answer to a query, even in the presence of inconsistent data. We may also be interested in repairing the database, so that the inconsistencies may be removed entirely.

Repairing Inconsistencies

When presented with an inconsistent database, we might be able to repair it. The nature of the repair will obviously be context-dependent. The basic definition of a repair D' of a database D is that D' is a modification of D such

that D' meets the constraint set C.

Basic Principles of Repairs There are infinitely many such database repairs that can be performed. We add a further restriction; that the new database D' is changed only minimally from the original D. Making the minimal set of changes necessary to meet all constraints C has nice properties: if a database is consistent, then the minimal repair is to do nothing (D' is equal to D). Furthermore, we will always be able to find a repair to the the database, since we can make changes as needed to reach a consistent state, although in the worst case, it may mean emptying the database entirely. [M. 99]

With that in mind, it is sometimes undesirable to make the minimal set of changes, because that may simply be deletion of the offending tuple(s). While deletion may bring the database into a consistent state, it also may mean data loss, and that is generally not our first choice. We are therefore forced to keep our minds open to repairs that are not the minimal set of changes, but as minimal as possible without data loss.

We will now look at an example using table of employee salaries, Table 4. In this example, we show only a subset of this relation; columns “employee_name" and “salary" are depicted. The sole constraint is that each employee may have only one salary.

salaries	employee_name	salary
	J. Page	50 000
	J. Page	80 000
	V. Smith	35 000
	M. Stowe	75 000

Table 1: Inconsistent Salaries Table [BC99]

The constraint is violated for the user “J. Page"; he has two salaries (rows 1 and 2 in Table 4) - $50 000 and $80 000 (cluster size of 2). There are two possible repairs that we can make to this database, depicted in Table 2 and Table 3:

salaries	employee_name	salary
	J. Page	80 000
	V. Smith	35 000
	M. Stowe	75 000

Table 2: Possible Repair of Salaries Table (Option 1) [BC99]

salaries	employee_name	salary
	J. Page	50 000
	V. Smith	35 000
	M. Stowe	75 000

Table 3: Possible Repair of Salaries Table (Option 2) [BC99]

Rows 3 and 4 of Table 4 are unchanged (as this is a minimal repair), and the two options are to delete either the first row (Table 2) or the second (Table 3). In either case, we return to a consistent database, because the sole constraint is now correctly enforced. Bertossi and Chomicki clearly demonstrate in [BC99] an important point: throwing away all tuples that participate in the violation results in the loss of J. Page’s data. They overlook the minor point that doing so would not be the minimal set of changes to the system - throwing away one would be enough.

There are some specific circumstances under which we might face infinitely many or infeasibly many repairs; such as how to repair a string such that it is unique in the relation. The number of possible strings which meet this condition is infinite (though it may be constrained from infinite to infeasible by limiting the string length attribute (for example, to 255 characters). Under those circumstances, the repair we undertake is to insert a `null` into that attribute, and therefore compress the infinite or infeasible set down to one possible repair [BC99]. The proponents of this system acknowledge that it is imperfect, however, because the relational data model may forbid a null in that attribute.

Specifying Repairs with Annotations The pressing question now is what repairs to perform. Annotated logic, specifically, *Annotated Predicate Calculus* as in [BC99], provides a method. We must move beyond classical logic, otherwise inconsistencies (logical contradictions) would prove problematic. In APC, we annotate database atoms (components) with true (t), false (f), contradictory ($\top$), and unknown ($\bot$). Below in Figure 1 is the lattice of logical values. In this figure, the intersection points represent truth values (t, f, et cetera) and the edges represent paths along which reasoning moves when additional data is found.

The truth values in the lattice are as follows, all from [BC99]:

- *Basic Values*: t, f, $\top$, and $\bot$, as explained in the preceding paragraph.
- *Database Values:* t_d for atoms in the original database and f_d for those not in it.
- *Constraint Values:* t_c and f_c to annotate database literals appearing in the disjunctive normal form of the constraints. Built-in atoms in the constraints receive simply t and f.
- *Advisory Values:* t_a and f_a are used to resolve situations where the database does not conform to the integrity constraints.

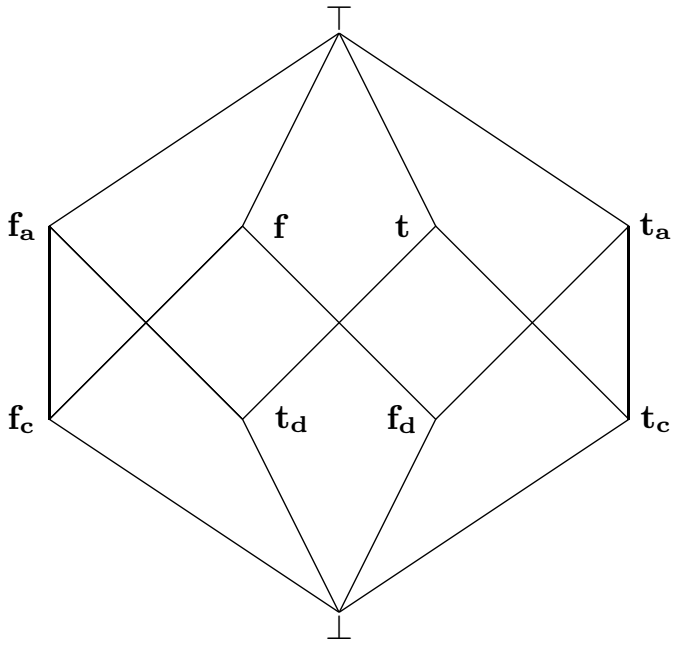


Figure 1: Truth Value Lattice [BC99]

Using Figure 1, we provide some clarifying examples. In the simplest case, if we have something true in the database (t_d), and true in the constraints (t_c), then we can move simply to true (t), without problem. If we find something true in the database (t_d) yet false according to the constraints (f_c), then we can move to f_a - suspected false. If we have an irresolvable conflict, for example, finding one thing certainly true and the other certainly false, we move to $\top$ (showing there is nothing we can do to resolve this).

We consider data to be fallible, but constraints to be correct. Thus, if an atom gets both t_d and f_c (true in the database but not permitted by the constraints), we will resolve that conflict as f_a; we think it should be false. At the end of the evaluation, in order to repair the database, things we believe to be false will be removed, and things we believe to be true will be inserted into the database. Bertossi and Chomicki in [BC99] assert that for every repair to a database D with respect to constraints C, there is a one-to-one relationship with a model of a minimal set of atoms tagged f_a and t_a.

If we apply this principle to an inconsistent database, then the results we will get will indicate our best course of action. The results received will simply be a set of advice: add this; delete that. A program can use the set of advisory values to repair the database.

Specifying Repairs with Logic Programs A reference in [BC99] presents a way to create a disjunctive program out of the set of advisory clauses produced using the annotated logic method. This program is used to process that advice and produce a repaired database. However, the program uses the annotations as additional data points, and it is therefore not an annotated (i.e. standard) logic program. Such a program will work for arbitrary first-order queries and for arbitrary constraints.

We can also make use of a logic program alone to find repairs. In [BC99] we consider a logic program with exceptions to these rules. The exceptions naturally have a higher priority than the rules. Most of the time, when we repair a database, most data is preserved, but some tuples will be removed. We can combine the appropriate logic program with the idea of logical consequence, and use this to compute repairs. To do so, we must find atoms true in every answer set of the program. This approach functions for all first-order queries, but it may be impractical because the size can quickly get out of hand.

Aggregate Queries Aggregate queries are important to databases. Basic operators like `MIN`, `MAX`, and `SUM` are valid queries and should not be excluded. It might be possible that some such queries could be expressed as a first order query, but in [BC99] it is acknowledged that we cannot handle such things using query transformation.

In some circumstances, the answers to aggregate queries are easy. If we look once again at the examples in Tables 4, 2, and 3. As an example, we want to select the minimum salary. V. Smith's salary of $35 000 is returned under all possible repairs. If instead we sought the maximum, both repairs offer different answers. The solution Bertossi and Chomicki present is that we cannot return a single answer. Instead, we return an interval where the minimum is the lower bound from all possible repairs, and the maximum is the upper bound. In this example, we consider [75 000, 80 000] to be a consistent answer to the aggregation query [BC99].

Open Issues The major open issue they identify is how to deal with global constraints in a composite database system. It is unclear what a consistent answer is in such a context, and similarly unclear what a repair means [BC99]. Bertossi and Chomicki further suggest that they are uncertain about how we can query such a composite system.

The final suggestion they put forward is to move this mechanism into the database management system. This would allow soft integrity constraints (not explicitly enforced), and different users could have different constraints, should they so choose. Under these circumstances, different consistent answers are returned depending on the query's origin.

Query Transformation

Bertossi and Chomicki presented another option: query transformation. Instead of modifying the database and then running the query on the new database, we transform the query and run it against the unaltered database. We change the query such that for every possible database D' the answers to said query is equal to the consistent answers (in light of constraints C in the original database D).

As in [BC99], we will consider first-order queries, and only universal integrity constraints. We iterate the transformation of the query by conjoining the *residues* to the results of the query, until the query in iteration n is identical to that in $(n-1)$ [BC99]. If there are no residues, then the query needs no alterations.

Residues But what is a residue? Consider the following example constraint set from [BC99]:

$$C = \{\forall x.[R(x) \bigvee \neg P(x) \bigvee \neg Q(x)], \forall x.[P(x) \bigvee \neg Q(x)]\}$$

We will focus on $Q(x)$ as our target query. The residue of $Q(x)$ with respect to the first constraint is $R(x) \bigvee \neg P(x)$; if the constraint and $Q(x)$ are true, then the residue must also be. The residue with respect to the second constraint is $P(x)$. Finally, the $\neg Q(x)$ component does not have any residues, because the integrity constraints do not constrain it. It is not necessary in this example, but in general, we might need to iterate the procedure, because the residues might themselves have residues.

Instead of running the query $Q(x)$, we ask $Q(x)$ together with each of its residues. More formally, it is $Q(x)$ is logically AND-ed with each residue, as follows:

$$Q(x) \bigwedge (R(x) \bigvee \neg P(x)) \bigwedge P(x)$$

The effect of this altered query is to return only the answers to $Q(x)$ for which the constraints hold. Looking at the previous example of Table 4, the modified query will ask for the names and salaries of all employees ($Q(x)$) where there is only one entry in the relation for that user (the residue, derived from the uniqueness constraint). This will return two tuples (V. Smith and M. Stowe). No data will be returned for tuples where there is a violation (J. Page), so no data is shown reflecting our uncertainty about his salary.

Algorithmic Validity Arenas, Bertossi, and Chomicki acknowledge that the algorithm hangs on three fundamental principle: soundness, completeness, and termination. While these are not examined in [BC99], they reference [M. 99], and all three principles are examined therein.

The soundness principle is the most obvious - every answer to the altered query must be a consistent answer to the original. The soundness principle is shown for all universal or non-universal but domain-independent queries. However, this excludes non-universal yet domain dependent queries like $\exists x \neg P(x)$ [M. 99].

Completeness is necessary so that every consistent answer to the original query appears as an answer to the transformed one. They prove completeness for binary and generic constraints, but do not prove it in general. In fact, in the event of disjunctive or existential queries, this method may fail to produce all answers [BC99].

Finally, termination means that there will always be a reachable point at which we may stop the query modification iterations because the query no longer changes between iterations. Fortunately, as long as the constraints set is acyclic, then termination occurs for any kind of query [M. 99][BC99].

Open Issues If we apply query transformation to the initial database in Table 4, this will certainly return only the consistent answers from the database (V. Smith and M. Stowe). Neither of the tuples for J.Page will be returned, because they are inconsistent. This may sometimes be desirable, if we do not wish to report information we are uncertain about. However, under some circumstances, it is probably undesirable to simply ignore the tuples that are inconsistent. Surely J. Page would prefer that the discrepancy in his paycheque be alerted to someone rather than have his information not shown in the accounting department's reports. To that end, he would almost certainly prefer to take home the lower amount than nothing at all. Query rewriting is useful, but not a complete solution.

The analysis in [M. 99] shows that query rewriting fails in the case of a disjunctive or existential queries, but only fails in the completeness category. The methodology is intended only to handle universal integrity constrains, and therefore existential qualifiers present problems. Arenas, Bertossi, and Chomicki assert that existential qualifiers may lead to co-NP-completeness, and computational feasibility issues prevent the fulfillment of the completeness criterion.

32 — Recovery: Repair, Probability

Probabilistically Answering Queries

Residues and repairs are not the only way to return consistent answers. When there are several options, as in Table 4, we can examine these options and make a determination of which is more likely. We form candidates - pretenders to the throne of the correct database - by breaking up the possibilities for repair into all of their possible variants; in each candidate database, one tuple from each cluster is selected. We see that in the example of Table 4, both candidate databases will receive a probability of being the “correct" database.

salaries	employee_name	salary
	J. Page	50 000
	J. Page	80 000
	V. Smith	35 000
	M. Stowe	75 000

Table 4: Inconsistent Salaries Table [BC99]

Finding Consistent Answers Andritsos, Fuxman, and Miller in [AFM06] take the approach of examining various probabilities of each tuple in a cluster being the correct answer. In the simplest solution, the database will just answer queries and attach the probability of the answer's correctness as another attribute of the tuple. Referring back to the salary example, below in Table 5 is Table 4 modified with the respective probabilities of each item:

salaries	employee_name	salary	probability
	J. Page	50 000	0.1
	J. Page	80 000	0.9
	V. Smith	35 000	0.4
	M. Stowe	75 000	1

Table 5: Inconsistent Salaries Table with Probabilities [BC99] [AFM06]

If the query being asked were all the names of all employees making more than $70 000, a probabilistic assessment would be performed to decide what certainty we could indicate the answers with. In the trivial case, M. Stowe's salary is certainly greater than $70 000, since its probability is 1 (completely certain). Uncertainty enters the picture when examining the J. Page tuples. We note probability of his salary being $80 000 is 0.9, so we include him in the return set and indicate the attached probability. If the query were names of employees making more than $45 000, then we would return J. Page with probability 1, since the sum of the probabilities of the tuples wherein his salary exceeds $45 000 is 1.

Andritsos, Fuxman, and Miller simply modify the requested queries to include the probability attributes. If the original query was:
`SELECT s.employee_name FROM salaries s WHERE s.salary > 70 000`
then the only changes are the addition of`SUM(s.probability)` and `GROUP BY s.employee_name`, so that the rebuilt query reads:

```
SELECT s.employee_name, SUM(s.probability) FROM salaries s WHERE s.salary > 70 000
GROUP BY s.employee_name
```

In a more complex query, we simply multiply the probabilities. Consider the following query [AFM06]:
`SELECT o.id, c.id FROM order o, customer c WHERE o.cIdFk = c.id AND c.balance > 10000`
This query is modified in the same way as that of the preceding paragraph, except the sum statement reads `SUM(o.probability * c.probability)` and the group statement is `GROUP BY o.id, c.id`.

Determining Probabilities The most pressing question is how we determine the probabilities. We note there are definite conditions attached to being able to determine these. See Figure 2 for the detailed breakdown of the procedure to map the tuples' probability to the interval [0, 1].

Input : A set of tuples $\mathbf{T}$,
- a clustering $\mathcal{C} = \{c_1, c_2, \ldots, c_k\}$ of $\mathbf{T}$,
  where c_i is the identifier of cluster i
- a distance measure d.

Output : For every tuple $\mathbf{t}$ in $\mathbf{T}$, a probability $prob(\mathbf{t})$.
Main Procedure :
- (Step 1) For $i = 1 \ldots k$:
  * compute cluster representative rep_i for c_i
    by merging all the tuples that belong to it.
  * initialize sum of distances for c_i, $S(c_i) = 0$.
  - (Step 2) For each tuple $\mathbf{t} \in \mathbf{T}$ that belongs to c_i:
    * compute $d_{\mathbf{t}} = d(\mathbf{t}, rep_i)$, the distance of $\mathbf{t}$
      to the representative of its cluster.
    * Add $d_{\mathbf{t}}$ to $S(c_i)$.
  - (Step 3) For each tuple $\mathbf{t} \in \mathbf{T}$ that belongs to c_i:
    * compute similarity $s_{\mathbf{t}} = 1 - \frac{d_{\mathbf{t}}}{S(c_i)}$.
    * $prob(\mathbf{t}) = 1.0$ if $|c_i| = 1$, or
      $prob(\mathbf{t}) = \frac{s_{\mathbf{t}}}{|c_i|-1}$ otherwise.

Figure 2: Tuple Probability Assignment Formula [AFM06]

Though most of the algorithm is self-explanatory, some components merit clarification, also provided in [AFM06]. Tuples within a cluster must be exclusive events, or the algorithm will fail. Representatives are determined by commonality. Representatives contain the most common features in the cluster, so the similarity between the tuples will give an indication of which tuple is most likely to be representative. To clarify, a short example, Table 6:

customers	name	market_segment	country	address
	Mary	building	USA	123 Jones Ave.
	Mary	banking	Canada	123 Jones Ave.
	Marion	banking	USA	123 Jones Ave.

Table 6: Inconsistent Customers Table [AFM06]

When examining this with a human's eyes, we might conclude that the most common values in the database are probably the correct ones, so the resultant representative tuple would show the customer Mary as part of banking in the USA with an address of 123 Jones Ave. The algorithm shown in Figure 2 follows this same intuition. The tuple we will consider correct is the one closest to the representative. For numerical data, similarity between two figures can be computed, and a pair can be more or less similar than another pair of figures (456 and 385 are more similar than 750 and 385). For data for which there is no obvious distance measure, we term them *categorical data*, and we proceed using information loss as the distance metric [AFM06]. Information loss is simply a measure of the difference between the tuple in question and the representative.

Analytical Shortcomings The strategy presented is imperfect, however, since we might fail to produce clean answers for some classes of query. Andritsos, Fuxman, and Miller present in [AFM06] the following case as unable to succeed: `select c.id from order o, customer c where o.quantity < 5 and o.cIdFk = c.id and c.balance > 25 000`. This fails because the join between `c` and `o` incorrectly double-counts some probabilities. Obviously, query rewriting cannot succeed for all cases, so the algorithm is limited to only those which we can reliably rewrite. Andritsos, Fuxman, and Miller argue – without proof – that such un-rewritable queries occur only infrequently, and that the algorithm is still valid most of the time.

Overlooked in this analysis is determining how we might find a representative sample when it is not obviously in a majority-rules scenario. Given two prima facie equally plausible tuples, how do we decide which is the better choice to consider the representative? Whichever is picked to be the representative will not differ from the representative (by definition), so it will receive a probability of 1. Thus, whatever we (perhaps randomly) choose to be our representative is the eventual winner and will be considered correct. It is clear that we need to come to some decision, but reporting 100% certainty about data which is, at best, 50% certain is misleading.

This analysis also equates popular with correct. If an incorrect answer appears in the database twice and a correct answer once, then the incorrect answer will be chosen and deemed correct, since it is more popular. That aside, there is not much that can be done about this problem; even a human observing the database might be more likely to conclude that the popular answer is the correct one, in the absence of additional, external knowledge.

Computational Complexity of Repair & Probability

Computational complexity is broken down into a short analysis on each of the methods detailed above. Each approach has its own properties. Computational complexity may disqualify a method from being practically useful, should it take too much time to reach a reasonable answer.

Repairing Databases Assuming that we are looking at repairs that are subsets of the original database, then repair checking is in polynomial time, for arbitrary constraints combined with acyclic dependencies [BC99]. Should any of these constraints not hold, the problem is pushed into the realm of co-NP-complete problems, although additional orthogonal restrictions might lead to more polynomial time cases [BC99].

Having considered the spectre of computationally infeasible or impossible situations, Bertossi and Chomicki [BC99] develop various methods of finding consistent answers without explicitly computing every possible repair to the database. One of the ways that they suggest is compact repair representation - use information present to construct an efficient representation of all the possible repairs, and use this representation to answer all queries. However, this is not explored in the paper.

Query Transformation The process of query transformation in [BC99] is shown to have a polynomial time computability of result tuples, as the transformed query will be first order as long as the original query is as well. In fact, because the query transformation does not require examining all the possible repairs, so we can evaluate a query with an exponential number of possible repairs in polynomial time.

Aggregate Query Transformation The approach presented in [BC99] is to aggregate queries is to build a conflict graph; a standard graph with nodes and edges. In the graph, maximal independent sets - that is, sets that are the farthest apart in terms of data equality - represent possible repairs of the database. As long as there is at most one nontrivial constraint, all operators except `COUNT` are polynomial time. If there are many nontrivial constraints, then the problem of finding lower and upper bounds becomes NP-complete. The `COUNT` operation is always NP-complete, even in the case of a single nontrivial constraint. Finally, it is noted that the `AVG` function's polynomial time algorithm is iterative, and cannot be formulated using SQL.

Under special circumstances, Bertossi and Chomicki found some better behaviour for the `COUNT` function, but even approximating is difficult; the concept of a maximal independent set reportedly has bad approximation properties.

Probabilistic Complexity Andritsos, Fuxman, and Miller include in [AFM06] a note on the computational complexity of their work, specifically a graph showing the performance as the size of the database grows, reproduced here as Figure 3:

It is clear from a cursory examination that the behaviour of the queries is linear, with the exception of some corner cases (like Query 3). However, they sidestep the issue that the number of candidate databases is possibly exponential - if there are n inconsistencies each with clusters of size m, the total possibilities might be as large as n^m. Computing probabilities for such a set might be exceedingly difficult, but the whole issue is glossed over with a single comment.

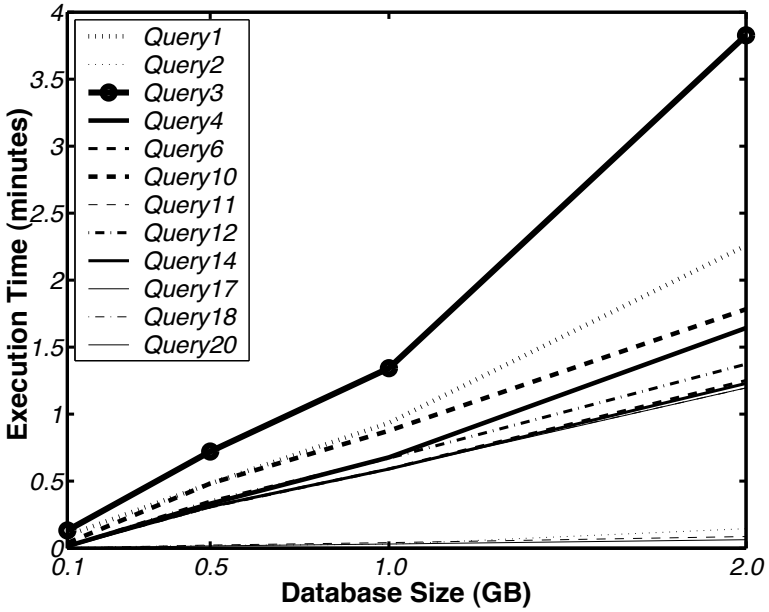


Figure 3: The Effect of Database Size on Probabilistic Evaluation [AFM06]

Future Work & Suggestions

In [M. 99], Arenas, Bertossi, and Chomicki warn that their proofs of termination and completeness properties of the query transformation are only preliminary and partial. Surely this will be addressed in the future. They also find that completeness for disjunctive and existential queries, they will need to move beyond simple query transformations. In addition, they do not yet have upper bounds on the size of the transformed query, and they lack some complexity information on different classes of query. This particular paper ([M. 99]) is a more detailed examination of some sections of [BC99]), with two authors in common, so suggestions for improving it are combined with that of its successor.

Database repairs examined in the [BC99] fashion are based on differences against the whole tuple and does not permit modifying attributes of tuples. Bertossi and Chomicki suggest that a *flexible* (attribute-level) repairs. They further identify majority-based approaches to consistency, which is very similar to how the probabilistic approach decides what to do.

Further in that paper, the section on logic programs to compute database repairs is rather light on details and seems mostly an aggregation of examples. We suggest that the paper might be improved by including more detail on the subject, since a lot has been left out, or reducing the amount of space dedicated to what is clearly a side point implementation detail.

In this same vein, there is a note about computationally constructing compact repair representations, yet there is not much talk about how we might do this, aside from a side note about placing `null` into a field when there are infeasibly many repairs. We suggest expanding compact repair representations, or clarifying the note about them (which implies that there is a wealth of information on the topic coming).

The probabilistic paper presents no future extensions or future directions to its work. As suggestions for them, they could certainly investigate the computational complexity further. Specifically, why some cases are more difficult than others to handle (see query 3 in Figure 3). They also appear to lack a strategy for dealing with a large set of inconsistencies.

Noting the scenario in which 50% certain data may be reported 100% certain, my suggestion for such forced-guess scenarios is to report the uncertainty associated with this answer and confess that we cannot find any better solution without additional input.

Finally, the set of queries that they cannot rewrite is claimed to be small, but no proof or evidence is offered to convince the reader that this is really the case. Surely a subsection or followup would be rather more persuasive than simply taking it on faith.

Conclusion

There appears to be no single right answer when dealing with an inconsistent database. However inconsistent it may be, it is unlikely that we will be able to unequivocally say what is correct and what is not. All of the examined options present viable alternatives to solving the problem, but they can easily slip into computational infeasibility in the event of complex constraints or a great number of inconsistencies. With these techniques, we can certainly get back consistent answers to our queries. The success of aggregate queries is less certain, but the chances are still reasonable.

Some techniques report uncertainties, and others ignore anything about which we are uncertain. Ultimately, how to handle the situations comes down to the decision of the database administrator or designer. He or she will need to choose the solution he or she feels appropriate to the situation.

33 — Parallel Databases

Parallelism in Databases

Although this is not a course on concurrency, parallelism, and synchronization, we will take some small amount of time to discuss how these impact on database implementation. Let's assume that we have a multicore system available because, let's face it, it's not 1999 anymore.

The first way that we could achieve parallelism or concurrency is based on how we decide to implement the server. If each incoming transaction to be processes is assigned a thread, then we could have n worker threads and a transaction gets picked up by a worker. Keeping in mind that locking and coordination are needed, but the maximum theoretical speedup is limited by the number of processors. Alternatively there could be a more segmented approach where rather than a worker that takes a job from start to finish there are different "segments" of a pipeline and those run in parallel. But that sort of thing is what we have covered in previous discussions.

I/O Parallelism

If the database is aware of the various disks available in the system, we can speed up performance via I/O parallelism. If we need to read some blocks from disk, we could let multiple requests run in parallel. If there are three blocks we need, and three disks, we could get all three results at (roughly) the same time.

We'll consider three basic partitioning strategies assuming we have n disks available [SKS11]:

- **Round-Robin**: When data is being written, the ith tuple is sent to disk i modulo n. This gives very even distribution of tuples.
- **Hash Partitioning**: Each tuple is passed through some hash function that produces a result in the range $[0, n-1]$ and each tuple of the relation is sent to the disk that matches the hash function.
- **Range Partitioning**: Assigns ranges of tuples to various disks. For example, is the values can be A, B, C, or D and there are two disks, we might assign A and C to disk 0 and B and D to disk 1 (or any other combination we like).

These techniques as described in the book ignore some other potential ways to decide, such as thinking how often tuples are accessed. Putting frequently used ones near one another might be sensible... or it might make more sense to put them on different disks so they can be retrieved in parallel.

Different approaches are good for different things. A quick recap from [SKS11] follows. There are three operations we consider likely to happen: scanning the entire relation, locating a tuple with some specific attribute equal to a certain value, and range queries (where we have a range of acceptable values of the attribute).

Round Robin is good for when we have to read the whole relation. But no advantage is gained when a specific query or range query is done as we have to search over everything anyway.

Hash partitioning is advantageous when we do specific queries. This is only beneficial, of course, if the items are partitioned based on that particular attribute. In theory the search time could be reduced to $1/n$ of the original time; otherwise no advantage is gained over round robin. This is also not especially suitable for range queries.

Range partitioning is good for specific as well as range queries on a particular attribute. A nice advantage of this is if we know that a query will be only on one disk, it's possible to send the query to just one disk and the other disks are available to perform other operations.

When the relation is spread over several disks, if things are not spread out evenly, then there is *skew* in the distribution of tuples: a large percentage of them being places in some areas and a very small percentage in others. This can be because of either *attribute-value* skew – that is, some values are more common in the data such as city being "Toronto" being much more common than "Yellowknife"; or it can be because of the way the partitioning function works even if values are evenly distributed [SKS11].

To balance this out, one suggested strategy is the histogram approach. Suppose there are, for example, 5 disks and 100 possible values. The simple range approach is 20 values in each partition. If the data has a normal distribution however (in the statistical sense of the normal distribution, the bell curve), rather than 20 values per disk we would try to cut it so there are about 20% of tuples in each partition.

An alternative approach is to cut it all up more. More? Suppose there are n processors and the work is divided up into $5n$ ranges. If the data is evenly distributed, each CPU will do 5 chunks. If the work is unevenly divided, however, some CPUs will do a smaller number of large chunks and others will do a large number of smaller chunks and some will be in between. The uneven distribution of values is then spread out over several processors.

Admittedly, in a single server database this is not likely to happen even when there are multiple disks. Usually the disks are arranged in a RAID array and that is done outside the purview of the database server... It is either managed by the operating system or by the hardware (even better). So why did we learn about this? Because it will be useful in a distributed database system, where there are multiple disks available to the database system. That is a subject we will return to soon.

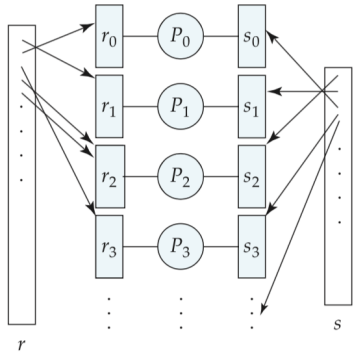
Intraquery Parallelism

The subject of inter-query parallelism is actually somewhat boring and redundant as it is just the same as previous discussions of parallelism and concurrency. Lock shared data, you know the drill. But there are numerous opportunities to get parallelism inside a single query, and it is a much more interesting subject. We'll consider two things: intraoperation parallelism (parallelizing an individual operation of the query) and interoperation parallelism (doing more than one operation at a time).

Intraoperation Parallelism. When we have a single operation that can be, in some way, divided up (partitioned) it is a good candidate for intraoperation parallelism. A number of them were likely things you already learned about in a concurrency course (just usually on some other data items), just now with tuples:

- Searching (e.g., linear search parallelized)
- Sorting (e.g., merge sort)
- Calculation (e.g., compute partial sums and combine)

Partitioned Join. To give a specific example, we will look at parallel join, in particular a partitioned join, as in [SKS11]. Let's assume we want to join relations r and s. If the join is an equality condition such as one attribute in r equalling another in s then we can divide this up to multiple processors. If there are n processors we could divide each relation into n pieces and send each processor P_i its piece of r and s. That processor then compute its part. These parts are then combined at the end to produce the correct data. Keep in mind that we do probably want to partition things with a histogram approach so each processor has roughly equal work.



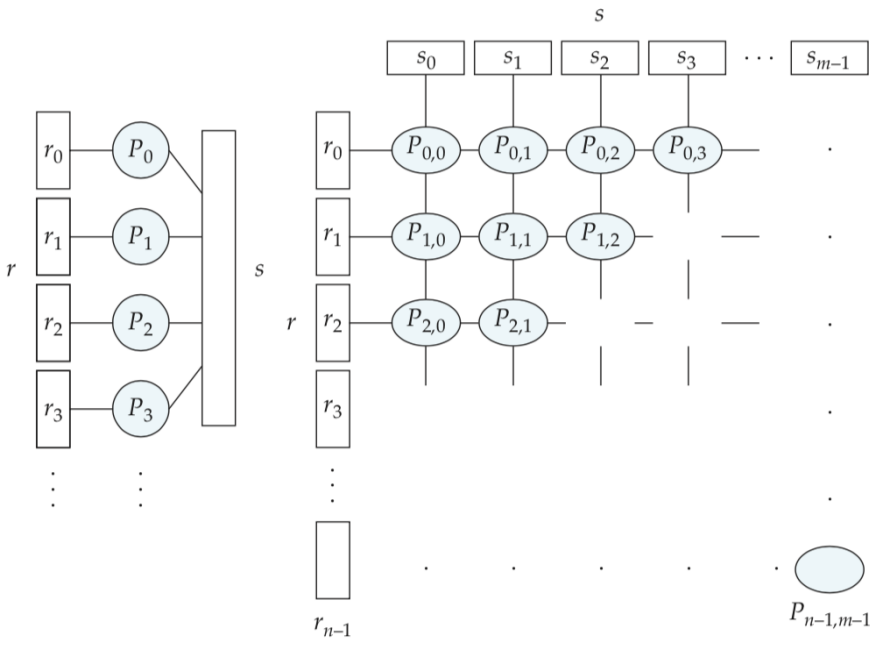
Partitioned Parallel Join [SKS11].

Fragment and Replicate. Partitioning is not suitable for all types of joins, however; because we put in the restriction that there is a join on an equality condition. A more general approach is the *asymmetric fragment-and-replicate join* which has three steps [SKS11]:

1. Partition one of the relations (r) and hand out the partitions to the processors.
2. Send copies of the whole other relation (s) to every processor.
3. Each processor computes its join (through whatever method) and the data is recombined.

It would be preferable to choose the smaller relation to be the one that is copied as it would obviously mean less data needs to be copied to execute this algorithm.

The diagram below shows the asymmetric fragment-and-replicate join; the general case appears on the right where there are $m \times n$ processors rather than just n. In a typical server, the asymmetric approach is suitable. But in theory if you had a very large number of cores then you might use the general replication routine.



Fragment and Replicate [SKS11].

A quick rundown on how parallelism affects operations other than selection and join [SKS11]:

- **Duplicate Elimination**: Duplicates are typically removed by sorting, which itself is something we could do in parallel. Also, if we partition a relation on an attribute, each processor can easily remove duplicates of that attribute.
- **Projection**: Projection can be done as each item is read from disk; just take what you need.
- **Aggregation**: Aggregation can be partially done locally and then the results can be combined to produce a global value. That is easy to imagine for something like sum (each processor computes its local sum, they are all added together). The same is true for count. Minimum and maximum involve comparison rather than addition to combine their values. Average, however, is more interesting: we would likely take the numerator (sum) and denominator (number of tuples) and recombine those.

Interoperation Parallelism. We already discussed the idea of pipelining: the partial result of one step can be moved on to the next stage and processing in that next stage will be able to begin processing. This, unfortunately, does not scale well: a pipeline with 4 stages cannot really take advantage of 16 processors.

The next way that we can use parallelism between various operations: if we are to join four tables, we could compute two pairs of joins in parallel and then combine them in a third operation for the last step. You probably will not want to join more than a few tables in any single query so the amount that this can scale is also limited.

Change is Hard

We would normally expect that when changes are made to the database schema such as data migration, changing column structure, or adding an index are made, the database is taken temporarily off the line (out of production) so the changes can be made while nothing else is going on. You have surely experienced this when you've tried to use an application on the web that tells you it's currently offline for maintenance. Such maintenance is usually planned for what are considered "off hours" (when utilization is low) and some downtime window is planned. Oftentimes when the downtime is significant it's because a database change needs to be made. And all things considered, when you have the option to do the change offline, you probably should.

Sometimes we don't have the option to take things offline for an extended period. The system is critical or there are other business reasons why you can't just turn things off. If the change is small it's not really a big problem. But we can do online migration if we must.

If a new index is to be constructed, for example, we can't just lock the relation in shared mode and we have to allow insertion, deletion, and update while the index is being added. This is usually done by just keeping track of all the changes as they come in and adjusting the index when generated to account for all the changes that happened while it was being built.

Parallel Database Architectures

Now if we have a parallel database, we should consider the architecture: how it is set up. We will consider three possible options: shared memory, shared disk, and shared nothing.

Shared Memory. This is our typical multicore or multi-CPU architecture we have likely already become familiar with when discussing concurrency. There are multiple processors (or multiple cores, processing units – as far as we are concerned all we want is workers). Memory is shared between the various processors and they have a common set of disks. There is no redundancy, however, as it is all wrapped up into one machine.

This means that communication can take place between the threads using shared memory. Every processor has its own cache and cache coordination is a factor, but the hardware helps us out there (for more details take ECE 459!). We should already be familiar with this sort of coordination from earlier: semaphores, mutexes, and the like, are used to achieve the coordination we need.

Shared Disk. In this architecture all processors can access disks directly, but every processor has its own memory. This has some degree of redundancy as we might be able to keep working if something goes wrong at one of the systems. If one of the systems crashes, every system can continue executing without any problem, at some reduced performance level.

The tradeoff is that communication has to take place using the disk. So to coordinate (e.g., lock an item) the lock information has to be written to disk and that means that disk access is likely to be a serious bottleneck in the system.

Shared Nothing. In a shared nothing system, well, it's exactly what it sounds like: nothing is shared between the systems. If they wish to communicate, then communication takes place over the network. This means the systems are as redundant as possible: if any one system goes down we might be able to carry on.

This sort of architecture isn't so much parallel as it actually is *distributed* and it comes with its own problems, such as what happens if we need data that is on the disk of some other system, since it does, after all, take longer to get data from a remote location than if it is locally available...

34 — Distributed Databases

Distributed Databases

A distributed database is basically a scaled up shared nothing system. The database is stored on multiple computers and almost always at multiple physical locations. The systems that make up the distributed database need not be the same (e.g., they may have different CPU types, different amounts of memory, different operating systems, etc). This is probably out of necessity: you can't order a computer with 4000 CPUs and 64 000 GB of RAM and 2000 hard drives, right? But you can order a thousand computers that each have 4 CPUS, that have 64 GB of RAM and 2 hard drives.

Our discussion of distributed databases is not in any sense a replacement for a distributed systems course but it might be a bit of an introduction to the topic that might increase your interest in it.

Transparency

When designing a distributed system one of the primary concerns is *transparency*, but not the sort we expect of governments where we should know what is going on. It's actually the opposite: what we want is for it to be imperceptible to the users that the database is distributed. There are a few types of transparency that are possible (and generally they are desirable but not necessarily) [EN11]:

- **Location Transparency**: The command used to perform a task is the same no matter the node on which it executes.
- **Naming Transparency**: When an item has a name, it can be accessed via the name regardless of location.
- **Replication Transparency**: Copies of the data can be stored in multiple locations for reasons such as performance, availability, or reliability. The user should not be aware that the data is a copy (which implies the copies need to agree... or at least not contradict each other).
- **Fragmentation Transparency**: The database can be broken up into multiple parts and it means a query may need to be broken up into multiple subqueries and recombined without noticing.

Fragmentation

Fragmentation is splitting up data. A relation r can be divided up into arbitrarily many fragments $r_1, r_2, ..., r_n$; combining all the fragments again will allow reconstruction of the original relation r.

Horizontal Fragmentation. In horizontal fragmentation, each part of the relation r_i contains some number of complete tuples. Each tuple is assigned to one or more fragments (but it has to be in at least one of them). If we were looking at a bank that had data centres in, say, New York and Hong Kong, then the tuples of the account data for the New York customers would be located in the New York data centre and the HK ones in HK. This keeps tuples close to where they are used the most, to minimize the data transfer requirements; it does not mean that the tuples are inaccessible at other locations; they are just going to take longer to retrieve [SKS11].

If it helps with understanding, the definition of the table is the same at both locations, but some rows are in one database and some in the other.

In relational algebra representation, a fragment r_i is created by a selection with a predicate. So a fragment $r_i = \sigma_{P_i}(r)$ and the entire relation can be reconstructed as a union of the sets [SKS11]. This works only if the sets are disjoint – no tuples are duplicated. If we have non-disjoint fragments then we might have the possibility of duplicate data, so we would need some sort of de-duplication to produce r again in a consistent way.

Derived horizontal fragmentation applies the partitioning of a relation rules to some secondary tables that are related by a foreign key [EN11]. Imagine you have a relation that is a shipment and there is a related relation of containers. Some number of containers are associated with the shipment (in the logic of the application); it would therefore make sense to split up the container relation based on their relationship to the shipment relation... If a shipment is at location 1, its associated containers should also be at that location.

Vertical Fragmentation. Vertical fragmentation is splitting up by attributes rather than tuples.

In relational algebra fragmentation is done using projection: $r_i = \Pi_{R_i}(r)$ and they are recombined using the natural join [SKS11]. For recombination to work, of course, we need a key in each part of the relation. If there are no attributes in common we have no way of recombining. Thus, at least some of the data will be repeated in the various partitions. In horizontal fragmentation the sets can be completely disjoint, but that is not the case in vertical fragmentation.

Now, the tables have different definitions – and there are some columns for each row in one database and some in the other.

Hybrid (Mixed) Fragmentation. This is the all-of-the-above sort of fragmentation, where both horizontal and vertical fragmentation are applied. Recombination of the relation is accomplished by performing the union and outer join operations in the appropriate order [EN11].

Replication

Replication is making copies of the data available in different database servers. This may be to improve availability or reliability of data. At the one extreme, all data is replicated to all sites in the distributed system, meaning every location has all the data. This has its advantages and disadvantages, as we'll examine.

The simplest way to replicate is to not do it at all: there are no duplicate copies of the data and whatever fragment is at each location is the only copy. This means to get the full data, all systems must be online and available.

If we do choose to replicate all of the data, then this is a fully-replicated distributed database. The system can continue to function as long as any one location is operational, and it can improve performance of retrieval of data since we can always query the closest database [EN11]. Unfortunately it means much more work on an update since all locations have to be updated. In fact, getting everyone to agree on what happened (or didn't) is a problem that is big enough to have its own minor topic.

In between is partial replication: some data is replicated over multiple sites and or all data is replicated in some way (e.g., any one piece of data is on at least three servers but may not be on all servers). A special case is when mobile workers go out into the field in areas where there is no internet connection, carrying a copy of the database, and have to synchronize when they get WiFi with some central server(s) [EN11].

Wherever data is replicated, we can choose one of the replicas to be the primary copy [SKS11]. If the whole database is replicated then one database site is the primary site and has the primary copy of all the data.

In databases under heavy load, replication is one of the keys to improving performance. A very common strategy is for reads to be done on replicas, with one database designated as the write master and all writes are done on the write master. The changes are then propagated to all replicas once the write is completed. This improves performance because we avoid the two problems we are going to talk about next: distributed locking and distributed transaction management.

Distributed Locking

If we take the single lock manager approach, there is one single lock manager and one site is chosen as the leader. All requests are sent to that one site and that system tells the other system whether the request is granted, denied, or delayed. A read can be done at any one site that has the data, but a write requires all sites that have that data to participate [SKS11].

The single lock manager approach is simple in that there are only requests and responses and everything is managed centrally so there is no need to invest effort to make sure that different lock managers agree. But there is the risk of failure at this one location causing the whole system to stop or misbehave.

If we choose instead to let all sites be involved in lock management, we have distributed lock management and the function is distributed. Immediately we can imagine this introduces the possibility that a disagreement arises about what data is locked and whether a lock can be granted. But it does mean that we no longer are constrained by the single site. We do have ways of handling the complexity.

Primary Copy. A simple approach to distributed locking is using the primary copy as controlling: if a site wishes to lock an item, it has to acquire the lock at the primary site. Thus, each site is responsible for keeping control over the data it is the primary for. The obvious downside is that if a site goes down, the data for which it is the primary is inaccessible even though there is a replica of it (until some other site takes over primary ownership) [SKS11].

Majority Rules. The next level up is to use a majority protocol: if a data item i is at n different sites, then a lock message has to be sent to (at least) $\lceil n/2 \rceil$ locations to lock i. If a site finds according to its own local situation that the lock can be granted, it returns a yes answer immediately; otherwise the response is delayed until the lock can be acquired. A transaction on i cannot proceed until a lock has been acquired on a majority of the locations [SKS11].

This routine requires a lot of messages: to lock an item requires $\lceil n/2 \rceil$ outgoing requests and also $\lceil n/2 \rceil$ responses; to unlock an item then another $\lceil n/2 \rceil$ messages.

There is also the possibility of deadlock in this situation. As you know, this occurs when there is a cycle in the resource requests. A transaction T_1 requests some item i_i and then i_2 and a transaction T_2 requests i_2 then requests i_1. Under normal circumstances we can just require that data elements have to be requested in some predetermined order (e.g., ascending order), that is to say, with lock ordering. Unfortunately for us, in distributed systems this is not a guarantee. A database can send out a request for i_1 then i_2 but because of the vagaries of the network the request for i_2 might arrive first. So we might need to rearrange some requests upon arrival or restart some operations.

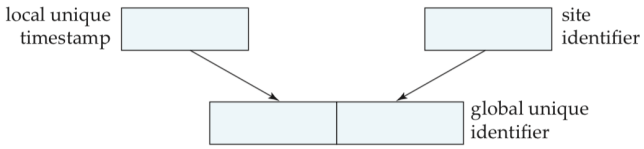
Biased Protocol. Rather than treating every lock and unlock the same, we could make a distinction on shared vs. exclusive locks. In the biased protocol, to get a shared lock, we just need to request a shared lock on an item from any one site that has a replica of that item. To get an exclusive lock, a lock is requested from all sites that have the item [SKS11].

If we expect that reads are more frequent than writes (this is very likely) then this is a performance increase. Reading any copy of the data can be done with just two messages, although a write requires $2n$ messages for each site where the data is stored. The request ordering problem is still at issue and the risk of deadlock still exists.

Quorum Consensus Protocol. In the quorum consensus protocol, every site is assigned a weight (non-negative) and then to perform an operation enough sites need to agree (that is, we must have a quorum). To execute a read, enough sites must agree such that their weights reach some threshold r; to execute a write enough sites must agree such that their weights reach a threshold w. The values for r and w can be the same or they can be set independently [SKS11].

Why would we choose sites to have different weights? We might have some sites that are remote and have poor internet connections, thus waiting for them might not be sensible. We may also have sites that have poor reliability, in which case we don't want to count on them to respond. In any case, choosing the values of r and w allow choosing how much agreement we need that locks have been acquired before we are allowed to proceed.

Timestamp Protocol. In the timestamp protocol, every transaction is assigned a unique timestamp. Each system uses its own local timestamp. Systems never agree on what time it is, so to make timestamps are unique, they are concatenated with the site identifier as shown below:



Generating unique timestamps in distributed databases [SKS11].

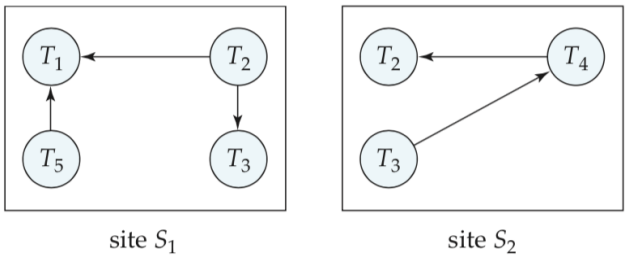
The placement of the site identifier in the concatenation is important; if we put it at the beginning then the timestamps at site k would always appear to be after those of $k-1$ and before those of $k+1$, which would not be correct.

A problem can still occur if the system timestamps are very far out of sync: if one site thinks it is one minute into the future it will always appear that its unique timestamps are after those of other locations. Fortunately, clock synchronization is usually a job of the operating system using NTP (the network time protocol). NTP allows the systems to synchronize their clocks in a reliable way; the protocol has been around for more than 30 years and we can realistically consider the problem (mostly) solved.

Lazy Replication. As a performance enhancement, databases may allow lazy replication: instead of having to do the write at all locations, we do write only at some location(s) and then the transactions get eventually transmitted to other sites.

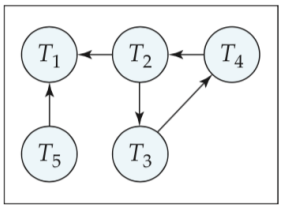
If we do updates at one location, the primary site, and they are eventually sent out to the other locations. Updates to an item are ordered serially because all updates to a single item take place at the same location; but other problems can occur such as an out-of-date value being used in a read as input to a write [SKS11]. The alternative is worse: if we allow updates performed anywhere and sent to all replicas then we can have concurrent updates of the same data at two locations.

Deadlock Detection. When we have a distributed database, a deadlock can occur but be difficult to detect using the wait-for graph algorithm, because nobody has the complete picture, as everyone has an incomplete wait-for graph. Each site may have its own local wait-for graph but that is not enough information to know if a deadlock has occurred (unless the local wait-for graph itself has a cycle). Consider the local wait-for graph situation shown below:



Local Wait-For Graph [SKS11].

With only a partial view, looking only at site 1 or looking only at site 2, we do not see any problem. But if we combine the information we have the graph below, which does have a cycle:



Global Wait-For Graph [SKS11].

Our approach for solving this is the centralized approach: one location is designated as deadlock coordinator and it is responsible for getting information from all other sites and aggregating it. The deadlock coordinator can then check this state and determine if there is a deadlock (and if so, use one of our resolution techniques to solve it).

It is noteworthy that the actual state of the system is not necessarily the same as the one seen by the deadlock coordinator; because there is a potential communication delay between the various sites so the information reaching the coordinator can be out of date [SKS11]. If the information is out of date we might accidentally signal a rollback on a transaction that is not deadlocked! Whoops. This can happen, but a transaction that is rolled back and restarted is really just delayed. We can live with a miscarriage of justice as long as our approach for recovering from this (e.g., restarting the transaction) is good.

Constantly checking the wait-for graph may be wasteful of resources if deadlock is rare. In fact, we might not even constantly send the wait-for graph information to the coordinator unless the coordinator asks for it. If a transaction has been waiting for a while, we might conclude that it has become stuck. That might prompt a site to send a message to the global coordinator asking it to run the deadlock detection algorithm. Or, the coordinator may choose to do so periodically, or when other conditions are fulfilled (e.g., some threshold number of locked data items has been reached).

Transaction Management

In addition to the usual transaction management, there is the global transaction manager that is, more or less, the boss of the other transaction managers. This may be a position where one database server holds the role permanently (it is the boss), or it might be that the database that handles the actual execution, or the one that started it.

Our biggest concern is making sure that everyone agrees. At a fundamental level that means everyone needs to know about the transactions, and needs to agree about whether the transactions are committed or aborted. The first strategy we will examine is the two-phase commit protocol [EN11]. A transaction is sent out to all the participating databases and they will do it.

In phase one, all databases signal the coordinator they are finished with actual execution. When the coordinator has received that from all participating databases, it sends out a message telling all of them to prepare for commit. Then all the databases force write all log records and anything else the recovery process demands to disk. If all goes well they send back a signal saying OK; if something goes wrong they send back a signal that says “Not OK”. If a timeout is reached without any response then a “Not OK” is assumed.

In phase two, if everyone said OK, and the coordinator also says OK, it sends a commit message to all the databases. Every database then proceeds with the commit. If something else happened, then the abort message is sent and all systems roll back the transaction.

At the end of this, all systems have either committed or aborted the transaction and a consistent state is ensured whether the transaction succeeds or not. If a failure occurs during step one it usually results in rolling back, but if a failure occurs during phase 2 it usually results in commit [EN11].

This protocol is good but it has some drawbacks. It is a blocking protocol; if the coordinator system fails, all the participating databases end up waiting until the coordinator system is back online. In the meantime they are locking data elements. Alternatively, if a coordinator and participant that have committed crash together, then we might get an indeterminate result [EN11].

The proposed solution is the three phase locking protocol (MORE PHASES!). The third phase is added in between

phase 1 and what previously was phase 2. The new second phase is that the coordinator shares with all participants the results of phase 2. Then phase 3 works just as the commit phase in the two-phase protocol. If the coordinator crashes now, then some other participant can step up and take the coordinator role and see the transaction through. Thus, we can be sure the transaction will finish no matter what system crashes [EN11]. This limits the wait time: based on the phase 1 information we know how the other systems will behave and we can just go ahead and commit; if no such message has been received, abort and proceed with other operations.

35 — NoSQL

NoSQL

NoSQL, well, originally "non SQL" or "non relational", is about data storage that isn't your typical relational database. Now it's "Not Only SQL", because some of the alternative databases can operate on SQL or SQL-like query languages (but that's sort of not what they are for). These days, NoSQL has become very popular and I sometimes have Fourth Year Design Project groups come to me and tell me they want to use NoSQL on a single user app that runs occasionally... Does that make sense?

If you ask them their motivations, such students will say something along the lines of, NoSQL is faster. As we will soon see, it can be but nothing comes for free. In any case, how much any particular app needs speed is an open question. Speed isn't everything.

In reality, the right place to start is assuming you want a relational database, that is, a typical SQL based system. There are a lot of advantages to this and they tend to scale pretty well for a typical workload. We can scale them up and we can scale them out, as we have discussed in the distributed databases section. As we proceed through this discussion we'll see what the tradeoffs are and why we might be willing to give up the advantages of the relational database because we, more or less, have no choice.

Motivation. The primary motivation for wanting to go to NoSQL would likely be scalability. That is to say, storing very large amounts of data, handling heavy workloads, and making data accessible to users wherever they are. When we say a lot of data we're talking about things like Twitter, Facebook, Uber, and other services that have astounding numbers of messages and millions if not billions of users (or, at least user accounts... there is a distinct possibility that a large fraction of Twitter and Facebook accounts are actually bots).

An upside of NoSQL databases is that they scale horizontally quite well. Unlike in SQL there is no need to do some magic to get a single server as big and fast as possible or to manually scale it across multiple servers. The MongoDB (one of the NoSQL vendors) documentation will tell you that one of the major advantages of the NoSQL approach is auto-sharding: no need to set up different database instances, get them to talk to each other, and the application probably needs be modified to work with multiple database locations. NoSQL, however, allows automatic sharding: they natively and automatically spread data across an arbitrary number of servers.

Let's imagine that you do have an application that really does have huge amounts of users or operates over a lot of user data, or both (e.g., Amazon). Users do both reads and updates: they look at products on Amazon and products can be purchased and can also be sold out. Doing something like NoSQL might let you do it and have higher speed and higher availability, but there are costs.

Tradeoffs. What we want to do is ultimately limited by the CAP theorem, also known as Brewer's theorem, which says that there exists an iron triangle in distributed data stores. The iron triangle, you might remember from a discussion about project management, is "fast, cheap, good: pick two". The CAP theorem says that it is impossible to get more than two out of the following guarantees: Consistency, Availability, Partition Tolerance [GL02]. To define these things:

- **Consistency**: Every read receives either the most recent write, or an error.
- **Availability**: Every request gets a non-error response, with no guarantee it contains the most recent write.

- **Partition Tolerance**: The system can continue even if messages are lost between nodes.

The iron triangle is actually perhaps not the best analogy. It's not as if, at the start of the system or even during the design phase, you just choose what two items you want. No, actually, instead of 2 out of 3, it's choose either availability or consistency (maybe the descriptions gave you this hint). This is because there *will* be network failures or at least delays, meaning there is partitioning, even if it is temporary. Then it's a question of what happens: if reads can happen before all nodes are updated, we get availability; if systems require locking all nodes before allowing a read, we get consistency [Mes13].

Let us consider two quick examples in the context of booking a flight. If we choose to prioritize consistency and there is network partitioning, then we would refuse to allow any updates like seat selection until such time as all servers are in agreement and the seat can be locked on all nodes. If we choose availability, then in the event of partition we allow the person to choose the seat anyway, which may be based on stale data. In that case, you might have let two people choose the same seat and there will be a need to sort this out later. For something like booking flights we might think it sensible to choose consistency over availability, because there is only one seat 32D on the airplane and it isn't exactly the same as any other seat. What if instead it was online shopping and you want to buy a book? One copy of the book is as good as any other, isn't it?

To give a real-life example of prioritizing availability over consistency, consider an online shopping scenario that really happened to me in November 2017. I was looking to buy a shirt and sweater, both of which were listed on the website as being available in the colours and sizes that I wanted. After suffering through the insufferable checkout process I received an e-mail telling me that I had successfully placed the order and the items would be shipped. Then, a while later I received another e-mail telling me they don't actually have the items and can't ship them and they're really sorry. And their website still showed the items as available. This is slightly embarrassing for the company, but it does happen.

For the most part, NoSQL databases do not provide ACID transactions. What you get is sometimes called BASE: Basically Available, Soft State, Eventually Consistent [Way11]. The first two points are fairly obvious, but the new part in that acronym is *eventual consistency*.

This is to say that eventually, after some nontrivial period of time, the data will reach consistency. This is something you might have experienced (more or less) in applications like Facebook; if someone posts something you may not see it eventually as it could take some time for that thing to propagate its way through the network to you. Now, the problem is you might never catch up, because the content is constantly being updated. People are posting new pictures of their dinner at every moment. Even so, that's not really a problem because the information will get to you eventually, even if you are constantly trying to catch up, you will never be THAT far behind, and also, it is not super important whether or not you learn whether your friend Terry ate that artisan heirloom radicchio.

As a developer it can feel like freedom to no longer have to worry about designing database tables and worrying about adding columns and thinking about foreign keys. The rules are just slowing you down, making you do a bunch of boring stuff, forcing you to talk with a database administrator about adding a column here or changing a data representation... Although there are probably some rules in organizations that you can do without, there are reasons for some of them that are valid and should probably be observed.

Structure helps in a few ways. It standardizes data representation where possible. If there is an address table in the database and different developers want to use it in different scenarios, they can re-use the already existing structure and you don't get three different implementations, all subtly different ("zipcode" vs "postalcode" vs "postcode"...). Moreover, if you make some changes in a SQL database to a table you will need to think about the migration strategy: if a data type will be shorter or a new field will be added or a structure changed then some way of converting the data from the old format to the new is needed. And this problem can be solved in NoSQL, of course, by doing it in the application: when you load a record, update it to the newest format, perhaps? Or write something to crawl through the data to update all the elements that need changing? In a relational database you're kind of forced to think about it because the alter-table statements may make it obvious that default values are needed.

Some of the NoSQL databases get speed by deleting some features that were very important throughout the course. There may be no joins. Really, no joins at all. And why would there be if there are no tables and therefore no good ways to relate two data elements together? If there are no joins then related data has to be stored together, namely in de-normalized schemas. This is exactly what it sounds like: it's a schema that is not in one of the normal

forms that we discussed. On the contrary, data is duplicated and things that should probably be separated are kept together. Because it's fast, you see; joins are slow so let's not have joins!

Thing is, though, joins enforce consistency, so we might lose consistency in our database as a result of this. Duplicate data, or multiple entries of the same data that are all slightly different (Jane Doe, Jane R Doe...) any one of which you might get back in a particular query. For some things you may not care all that much – if it is something like keeping track of how many megabytes of data the user has used this month then "close enough is good enough"... Maybe.

Some NoSQL implementations do allow joins but they don't work quite like your typical SQL join with join attributes defined in a query... You can, if you want, write your own sort of join, where you do some sequential lookup: first look up record x which contains a way to find related record y... This might be acceptable for finding an individual record, but to do so for 10 000 records is tedious. So perhaps you need to write your own join? At least we learned how those algorithms work...

But then, why are you trying to reinvent the wheel? Relational database vendors have poured a lot of time and money and effort into optimizing every part of the database engine, including joins and fetches and the like. Even sometimes strange things like trying to time your code execution with the position of the read arm of the hard disk drive to micro-optimize a read or write operation [Way12].

Another way that NoSQL might work is it might not have transactions at all. Things happen, or don't, or might be halfway completed. It's up to you as the application developer to try to do things atomically or have some sort of failure recovery (rollback) mechanism. We'll consider, later, a case study about the Oracle NoSQL product that has something resembling a transaction.

Types of NoSQL Databases

As you might already know, the variants of SQL that different databases speak can cause some problems: you can't necessarily take a query written for MySQL and use it in a Postgres database. It might be necessary to convert it because keywords are a little different.

Instead of working on the basis of relations with tables and keys and whatnot, how do NoSQL databases work? There's no simple, single answer to this because there are a lot of options. Remember, this is about things that are not standard relational databases. There are some options that support multiple modes, but we'll choose to focus on just a few things for now: key-value databases, document databases, column databases, and graph databases.

Nevertheless, NoSQL tools do not have any sort of standardized language between them. Every vendor has their own idea about what is best and how things should work. In some cases, the queries are expressed as functions or they can be written in a query language that has some similarities to SQL queries. As a result of this, the costs of switching between tools can be high; significantly higher than it would for switching between SQL database providers.

Key-Value Databases. Surely at this point you are familiar with the idea of a key-value pair: in data structures and algorithms you learned about a Map (HashMap) for example that operates on the basis of `get( key )` and `put(key, value)`.

Key-value pairs are a recommended way to store small amounts of data for things like Android applications. If you want user preferences in an app to be saved so it is remembered when you open the app again, this is one way to do it. And you can put arbitrary things in there, so you could save the configuration as XML or even just put the (serialized) data structure in there. As long as you know what it is and how to interpret it, you can get it out again. This is small-scale, obviously, but the idea has merit.

Now that idea is applied to a larger amount of data. Instead of defined tables with specific attributes, you have an arbitrary key and an arbitrary value. There may be no restrictions on the form of the key or the form of the value; they can just be bytes of any length with no specific format, rules, or limits. This allows you to store anything you like... pdf documents, Java objects, flat text, comma-separated data... anything.

This scales well, since the only operations are get (read) and put (write) and there are no complexities related to searching, foreign keys, transaction management, et cetera. Eventual consistency can be achieved when every

location, sooner or later, gets the update of the data. Put operations simply overwrite old data, and data elements (values) have no formal relationships to one another. You can create some ad-hoc ones by storing under one key a list of other keys, but this is informal.

Another thing that we don't get in this situation is aggregation or other "magic" the database can do for you. Remember that you can ask a relational database to sum a certain thing for you: how much income was generated in July of 2016? This is a select with aggregation and we can load just the data we need and crush it down to a single number (385 000) and transmit that result. In a key-value store, no such functionality exists. You can query the data that meets your certain criteria, but you have to do your own addition in application code. That itself isn't necessarily bad but your performance might be very slow if you have to load a large amount of data from the database and send it to the application. That communication cost can be killer depending on the volume of data, the connection type, connection speed, and so on.

Column Databases. Column databases are pretty much like key-value pairs with a slight bit of structure tacked on to the value part of it. In this case, a value has three components: name, the content ("value"), and a timestamp. The name is really the key; obviously, the value and timestamp are stored under that key. A column is then a logical view that encompasses, key, value, and timestamp all together. The timestamp is used if we ever need to differentiate between different versions of the data and in case of recovery.

Document Databases. A document oriented store is somewhat more structured than the simple key-value pair. Instead of just treating the value in they key-value pair as an opaque series of bytes, we might analyze it and do something with it. Based on the structure of the value, as a document, it might be possible to extract some (meta)data about the document and then use that for something else (e.g., search).

We could store lots of documents in the database in formats like XML, JSON, or binary data like PDF or Word documents. The documents themselves don't need to be formatted in any specific way, follow the same format or conform to any schema. And the content is completely arbitrary. If a document contains five XML tags, it has only those and there's no need to put nulls for XML elements that don't exist in that document. And, of course, no need to convert it from XML to an insert statement or convert the output of a select statement back into XML in the future. While there can be some tools that do that for you automatically so that you don't have to write the conversions manually, the work done is nonzero.

The basic operations are generally defined as CRUD [Mar83]: **C**reate (insert, put), **R**etrieval (query, search, get), **U**pdate (edit), **D**elete (remove). These can come under other names but this acronym is well known. These could be translated to get and put operations, but CRUD is a more friendly API that makes well to use cases.

Aside from the convenience of CRUD rather than get/put, the major advantage is the analysis that allows searching or perhaps better query performance. If we wanted to search for documents with a certain element, e.g., find all documents relating to a particular Tax ID, then having done the meta-analysis we could (maybe) find them relatively quickly if we have an index. If we are willing to have a bit more structure, we could allow differentiation of the data. Suppose the documents are e-mail and we have a bit of structure. Then if we want to search for "abc@xyz.com" we could find e-mails where that address corresponds to the recipient but not the sender.

A friend and database administrator referred at one point to a document database store as a "Datengrab", German for "Data Grave". It's where documents are placed when they die. If they ever come out it's nice, but admittedly in this particular use case, if the document happens to fall off a cliff it can always be regenerated from the source. This is, interestingly, a case where both things are used: there is a relational database that is used for most data, but documents are put into the non-relational document database. The choice is not binary.

Graph Databases. In a graph database, a graph structure is used: there are nodes, edges, and properties. Nodes and edges are first-class entities. What we would normally consider an entity in a relational database is modelled as a node. What is modelled as a relationship in the relational database is an edge. Nodes can have properties, and so can edges. This approach might even seem more natural for certain use cases than the standard relational model approach where everything is converted, eventually, into a table (or attributes within a table).

This supports a lot more ideas about joining, relationships, and constraints, while of course avoiding a lot of the structure imposed by the standard relational database. In fact, we have something that looks like the document store model – there are only two kinds of first-class elements (instead of everything being a table) – but we don't necessarily give up the idea of formal relationships between entities [Cox17].

Example: Oracle NoSQL

You have probably heard of Oracle, you know, those guys who make the crazy expensive, enterprise-grade relational databases. They made a NoSQL database, and, well, it's their way of doing things: intended for the enterprise. It is fair to say that the Oracle approach is somewhere between the standard relational database approach and the NoSQL no-rules, speed-is-everything, forget-safety kind of approach.

The Oracle NoSQL database provides something like transactions that are a practical approximation of ACID compliance. Instead of a simple key-value structure, Oracle divides the key into major and minor parts; the major is an object identifier and the minor parts are the "fields" in the record. So instead of just having the value as an impenetrable blob, it has an object with multiple fields and the names of those fields are now just called minor keys. So if a user has some unique ID and then associated things like addresses, the major key is the user ID and the minor keys are the address elements [Way11].

In short you get an ACID "promise" when all writes in a group are attached to the same major key. This makes it suitable for certain types of work: updating a user's personal information and storing it, for example. It is not suitable, however, for a standard bank example of transferring money from account A to B. Because the transaction hits two separate accounts, under two separate major keys, there is the distinct possibility that the transfer will not work as planned and moving \$50 from A to B does not result in the total sum of money remaining the same after the transfer. And while banks probably enjoy the thought of taking \$50 out of your account and not giving it to someone else, there are kind of sort of laws against that.

This promise is fulfilled by the implementation of their NoSQL database: one master machine is guaranteed to hold all minor keys associated with a major key [Way11]. That's actually super convenient because it means that all these minor attributes are on the same node and we can therefore get consistent behaviour through the use of local locks and it all works. But no such guarantee is provided for when the major keys are different because they could be located on different nodes. They might be on the same node and things might work, but there are no promises.

Oracle NoSQL offers both kinds of scaling, replication and sharding. With sharding, your data is spread out and you have less contention (faster writes). If you have replication you get faster reads and higher availability of data (and reliability in some sense). Of course, the more replication you have the longer it can take to write some data, because you do have to get all the systems to agree.

But even that is configurable: you can tell the system what "durability policy" you want to have. Choose if writing successfully to one node enough, or if a simple majority is sufficient, or if all nodes need to agree on the write. Values are associated with a version number which you can use in writing your own sort of replication and durability policy if you so desire [Way11].

Conclusion

NoSQL has lots of advantages and disadvantages when compared to standard relational databases. If used in the right situation, it can speed operations and provide a lot of scalability and availability. If used in the wrong situation it either wrecks your application/company, causes endless headaches, or forces you to reinvent the wheel and implement a lot of features that relational databases come with out of the box. Start with the idea of using a relational database, and consider carefully what you would be losing if you went to a NoSQL approach. If it's the right call for this need, then, by all means. You have the power and the choice is yours. Choose wisely.

Bibliography

[AFM06] P. Andritsos, A. Fuxman, and R. Miller. Clean answers over dirty databases: A probabilistic approach. *Proceedings of the 22nd International Conference on Data Engineering*, 2006.

[BC99] L. Bertossi and J. Chomicki. Consistent query answers in inconsistent databases. *Proceedings of the eighteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, 1999.

[Buc11] Craig Buckler. How to Create Triggers in MySQL, 2011. Online; accessed 14-November-2017. URL: `https://www.sitepoint.com/how-to-create-mysql-triggers/`.

[Cod70] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, June 1970.

[Cox17] Graham Cox. Introduction to graph databases, 2017. Online; accessed 30-November-2017. URL: `https://www.compose.com/articles/introduction-to-graph-databases/`.

[Duh12] Charles Duhigg. How Companies Learn Your Secrets, 2012. Online; accessed 17-October-2017. URL: `http://www.nytimes.com/2012/02/19/magazine/shopping-habits.html`.

[Ede17a] Lucas Eder. JOIN Elimination: An Essential Optimiser Feature for Advanced SQL Usage, 2017. Online; accessed 2-September-2017. URL: `https://blog.jooq.org/2017/09/01/join-elimination-an-essential-optimiser-feature-for-advanced-sql-usage/`.

[Ede17b] Lukas Eder. How modern sql databases come up with algorithms that you would have never dreamed of, 2017. Online; accessed 19-October-2017. URL: `https://www.slideshare.net/LukasEder1/how-modern-sql-databases-come-up-with-algorithms-that-you-would-have-never-dreamed-of`.

[EN11] Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems, 6th Edition*. Addison-Wesley, 2011.

[GL02] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, June 2002.

[Lev11] Frederico Leven. An Introduction to Stored Procedures in MySQL 5, 2011. Online; accessed 14-November-2017. URL: `https://code.tutsplus.com/articles/an-introduction-to-stored-procedures-in-mysql-5--net-17843`.

[M. 99] M. Arenas, L. Bertossi, and J. Chomicki. Consistent Query Answers in Inconsistent Databases. *Proceedings of the eighteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 68–79, 1999.

[Mar83] J. Martin. *Managing the Data Base Environment*. A James Martin book. Pearson Education, Limited, 1983. URL: `https://books.google.de/books?id=ymy4AAAAIAAJ`.

[Mes13] Lior Messinger. Better explaining the CAP Theorem, 2013. Online; accessed 28-November-2017. URL: `https://dzone.com/articles/better-explaining-cap-theorem`.

[Sao15] Oum Saokosal. Database Normalization 1NF, 2NF, 3NF, BCNF, 4NF, 5NF, 2015. Online; accessed 16-November-2017. URL: `https://www.slideshare.net/kosalgeek/database-normalization-1nf-2nf-3nf-bcnf-4nf-5nf`.

[SGG13] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts (9th Edition)*. John Wiley & Sons, 2013.

[SKS11] Abraham Silberschatz, Henry F. Korth, and S. Sudarshan. *Database System Concepts, 6th Edition*. McGraw Hill, 2011.

[Sta14] William Stallings. *Operating Systems Internals and Design Principles (8th Edition)*. Prentice Hall, 2014.

[Tan08] Andrew S. Tanenbaum. *Modern Operating Systems, 3rd Edition*. Prentice Hall, 2008.

[Way11] Peter Wayner. First look: Oracle NoSQL Database, 2011. Online; accessed 30-November-2017. URL: https://www.infoworld.com/article/2621199/database/first-look--oracle-nosql-database.html.

[Way12] Peter Wayner. 7 hard truths about the NoSQL revolution, 2012. Online; accessed 29-November-2017. URL: https://www.infoworld.com/article/2617405/nosql/7-hard-truths-about-the-nosql-revolution.html.

[Wen14] Kris Wenzel. Database second normal form explained in simple english, 2014. Online; accessed 15-November-2017. URL: https://www.essentialsql.com/get-ready-to-learn-sql-10-database-second-normal-form-explained-in-simple-english/.