Functions in PGPLSQL

functions in psql
create or replace function funcName()
return **s** returntype as **$$**
begin
raise notice 'Hello Birb!';
end;
$$ language langName
The two $ are always necessary. Also note the
returns with an s and the language at the end.
Which **MUST** be a **PROCEDURAL LANGUAGE**,
so c++ doesn't work here.

Parameters are handled like in any language
func(x bigint, y bigint)
you can also define multiple return types
func(variadic a numeric[])
or a generic return
func(param anyelement)

Variable Declaration:
returns void as $$
DECLARE
x bigint; y bigint;
BEGIN ....

Variable manipulation: x **:=** 6 + 4

if: IF n = 0 THEN RETURN 1;
(optional)ELSE RETURN 2; END IF;
ELSIF also possible (note elsif not elseif)

case x when 1, 2 then msg := 'one or two'; end case;
essentially this checks if x is 1 **OR** 2

case when x between 0 and 10 then ....
similar but with a range, both can be
simulated by if else.

Exceptions: BEGIN z:= x / y;
EXCEPTION WHEN division-by-zero
THEN z:= 0; (or error rather) END;
if you want to catch all: WHEN others THEN
often used after exception: RAISE; (show error)

For Loop: For var IN query LOOP
statements END LOOP;
for r in SELECT * FROM ang LOOP
RETURN NEXT r; END LOOP; RETURN; END;
note that the return next doesn't return
you store it in a buffer
and return it at the end of the function.
for infinite loops: FOR i IN 1.._max LOOP;

update and insert: INSERT INTO ANG VALUES(...);
UPDATE ang set salary = salary + 500
where name = 'dashie';
interestingly, after the where name = 'dashie'
you can use if not found then (handle error)
this allows for easier error handling.

queries: **execute** 'SELECT * from ang'
—— into result; return result; END;

comments are done by either – or /* */ for multiline

anonymous function: you can omit
the name and just write do $$ ...

cursor: declare curs CURSOR FOR query;
BEGIN OPEN curs; LOOP do something CLOSE
curs; END;
Cursors are essentially just iterables.
cursors can also be unbound curs1 refcursor
or they can be parameterized curs3 cursor(arg)

- **PL/pgSQL: Datentypen**
  - Boolean:
  - Zahlen: int, integer, number
  - Strings, Datum, etc.
  - Arrays: alle Datentypen gefolgt von „[]“, z.B. int[]
  - Weitere: JSON, etc.
- ... ergänzt mit zusätzlichen Datentypen:
  - var5 angestellter.id%type; -- abgeleiteter col.-Typ
  - var6 angestellter%rowtype; -- abgeleitet von Tabelle
  - var7 record; -- generischer Record `This is a simple entry inside a qurey aka 1 person`
  - var8 anyelement; -- generischer Typ gemäss Fn.-Argument, vgl. nachfolgend
  - curs1 refcursor; curs2 cursor ...; -- vgl. nachfolgend

arrays: SELECT '1,2,3'::int[]
or SELECT ARRAY[1,2,3]
var int[] only in variable declaration.
!!arrays start with 1 in psql !!

return types: all of the above AND void,
SETOF type (array of a type), TABLE, Trigger

Arrays: Accessoren

```
create table tictactoe as
  (select 1 as id,
        array[
              ['z1 k1', 'z1 k2'],
              ['z2 k1', 'z2 k2'],
              ['z3 k1', 'z3 k2']
             ] as board);
```

Index Query: intuitiv wie eine Koordinate ("1-basiert": Start mit 1 nicht 0):
```
select board[1][1] from tictactoe;
-- z1 k1
```

Slice Query: "Untergrenze:Obergrenze" für jede Dimension:
```
select board[2:3][1:1] from tictactoe;
-- {{z2 k1},{z3 k1}}
```

Max-Bound-Abkürzung "[2]" **vermeiden** (Verwechslungsgefahr), besser [1:2]
```
select board[2:3][2] from tictactoe;  -- [2]=>[1:2]
-- {{z2 k1,z2 k2},{z3 k1,z3 k2}}
```

Suche mit **ANY**:
```
select * from tictactoe where 'z2 k2' = any(board);
-- 1;{{z1 k1,z1 k2},{z2 k1,z2 k2},{z3 k1,z3 k2}}
```

Arrays: Operatoren

- «Is equal»: =
- SELECT ARRAY[1,2,3] = ARRAY[1,2,3]; true
- SELECT ARRAY[3,2,1] = ARRAY[1,2,3]; false

«Contains»: @>
- SELECT ARRAY[1,7,4,2,6] @> ARRAY[2,7];

«Is contained by»: <@
- SELECT ARRAY[2,7] <@ ARRAY[1,7,4,2,6];

«Overlaps»: &&
- SELECT ARRAY[1,4,3] && ARRAY[2,1]

hstore / map:
Create und Insert:

```
CREATE TABLE test (id integer, col2 hstore, col3 text);

INSERT INTO test VALUES (1, 'a=>123, b=>foo, c=>bar'::hstore, NULL);

SELECT * FROM test;
id| col2                                  | col3
---------------------------------------------------
1 | "a"=>"123", "b"=>"foo", "c"=>"bar"| null
(1 row)
```

Queries:
- List all keys
  SELECT akeys(mykvpfield) FROM ...
- Get all key-value pairs
  SELECT each(mykvpfield) FROM ...
- Get key value (as text)
  SELECT mykvpfield->'name' FROM ...
- Test if left hstore is contained in right hstore:
  ... WHERE mykvpfield @> 'tourism=>zoo'; -- or hstore('tourism','zoo')

Operatoren:
- "->" get value for key : SELECT 'a=>x, b=>y'::hstore -> 'a'
- „@>“, etc. ... ähnlich wie Array-Operator

hstore supports GIST/GIN indexing

	FUNCTION	PROCEDURE
Use in an expression	✓	✗
Return a value	✓	✗
Return values as OUT parameters	✓ (PG Spezialität)	✓ (PG v14)
Return a single result set	✓ (as table fn.)	✓
Return multiple result sets	✗	✓
Contain transactions	✗	✓
Make it run using ...	EXECUTE	CALL

also note that type%rowtype is used like this:
r ang%rowtype -> for r in select * from ang;

important to know, you can always use these
functions to manipulate queries, for example
select upper(name) from ang;
depending on the function you can also
select generatetab(1,10)

- Es gibt zusätzlich IN, OUT, INOUT
  `create function foo(IN p1 type)...`
  - IN: call by value; Variablen oder Ausdrücke als Argument
  - OUT: call by reference; nur Variablen als Argument
  - INOUT: beides

cast: cast(input as type);
cast(record.id as text);

stored procedures are nothing but a chaining of functions:
- Schritt 1 in PL/pgSQL: siehe Beispiel 3 (SP-Funktion mit in/out –Parametern)
- Schritt 2: Deklaration in Java/JPA (aka Registrierung in JPA):

```
@NamedStoredProcedureQuery(
  name = "MySum", -- JPA-Objekt
  procedureName = "mysum", -- Name der SP-Fn. (DB-Objekt)
  parameters = {
    @StoredProcedureParameter(mode = ParameterMode.IN, type = Double.class, name =
"x"),
    @StoredProcedureParameter(mode = ParameterMode.IN, type = Double.class, name =
"y"),
   @StoredProcedureParameter(mode = ParameterMode.OUT, type = Double.class, name =
"sum")
  }
 )
```

- Schritt 3: Call it!

```
StoredProcedureQuery query = this.em.createNamedStoredProcedureQuery("MySum");
query.setParameter("x", 1.23d);
query.setParameter("y", 4.56d);
query.execute();
Double sum = (Double) query.getOutputParameterValue("sum");
```

some good to know things: plain SQL is more efficient.
write variables lower case for sql
use cast over typename -> not select date '2022-06-07'

Triggers

- sind DB-Objekte und immer einer Tabelle zugeordnet
- werden in Stored Procedures programmiert
- haben keine Parameter
- können nicht direkt aufgerufen werden
- werden vom DBMS beim Eintreten eines Events aufgerufen
- haben bei der Ausführung die Rechte ihres Owners

- Trigger can pass parameters to function
- FOR EACH [statement | row]
- Events: INSERT, UPDATE, DELETE, TRUNCATE, INSTEAD OF
- Function executes BEFORE or AFTER changes

statement is once, row means once
per row, aka for the entire table.

- Before triggers can change contents of new row
- After triggers can only respond to what has happened e.g. Foreign Keys
- Return values of AFTER triggers will be ignored
- Triggers execute in alphabetical order

```
The INSTEAD Trigger can be use
to avoid crashes:
-> INSTEAD OF UPDATE
-> ON UPDATE DO INSTEAD
for example, trying to update
a read-only view -> INSTEAD OF
UPDATE
```

- DDL für Triggers
  ```
  CREATE | DROP | ALTER TRIGGER
    mytrigger …
    ON mytable …
  ```
- Syntax-Beispiel CREATE TRIGGER:
  ```
  CREATE TRIGGER mytrigger
    AFTER INSERT OR UPDATE
    ON mytable
    FOR EACH ROW
    EXECUTE PROCEDURE mytriggerfn();
  ```

PL/pgSQL: Trigger-Fn.-Variablen

TG_NAME	Name des Triggers (TG)
TG_WHEN	BEFORE oder AFTER
TG_LEVEL	ROW od. STATEMENT
TG_OP	INSERT, UPDATE, DELETE, (TRUNCATE)
TG_RELID	OID der Tabelle
TG_RELNAME	Name der Tabelle
TG_TABLE_SCHEMA	Schema der Tabelle

Syntax-Beispiel Trigger-Fn. passend zu Trigger:
```
CREATE FUNCTION mytriggerfn()
RETURNS TRIGGER --
AS $$
  <<body>>
$$ language plpgsql;
```
- Eine Trigger-Fn. hat keine Fn.-Parameter
- Diese werden über Trigger-Fn.-Variablen übergeben, u.a.:
  - TG_NARGS Anzahl Parameter
  - TG_ARGV[] Array von Parametern als TEXT

```
CREATE OR REPLACE FUNCTION dt_trigger_func()
  RETURNS TRIGGER AS $$
BEGIN
  IF (TG_OP = 'INSERT') THEN
    NEW.creation_date := now();
  ELSIF (TG_OP = 'UPDATE') THEN
    NEW.modification_date := now();
  END IF;
  RETURN NEW;
END
$$ LANGUAGE plpgsql;
```

BEFORE-Trigger : Falls RETURN null ➔ wird Operation abgebrochen

```
CREATE TRIGGER dt_trigger
BEFORE INSERT OR UPDATE
ON mytable
FOR EACH ROW
EXECUTE PROCEDURE dt_trigger_func();
```

Return types: RETURN NEW -> returns a new table/row
RETURN OLD -> returns the old table/row
(but could change other rows!)
RETURN NULL -> cancel operation.

running order: before statement, before row,
after row, after statement ->
and of course alphabetically.

inside the Trigger functions you can use the
variables that don't matter aka can be ANY -> user
or the entered user from the trigger -> NEW.user
or explicitly the old one -> OLD.user
and last user defined stuff like -> SELECT 'I'
which just places an I as the variable
or something like now() for timestamps.

Triggers make the database slower and harder to maintain.
some databases therefore let you disable them if you want.
On a table basis.
Also watch out for cascading effects of triggers
they might cause something else to be deleted.

Stored Procedures are really helpful for security
They have all the prviliges, but only
allow the user to do what the
creator has predefined.

User defined types in SQL

CREATE DOMAIN ...
- erstellt einen benutzerdefinierten Datentyp
- für Datenschema und Stored Procedures
- mit Constraints wie NOT NULL, CHECK

```
create domain contact_name as
  varchar(255) not null
    check (value !~ '\s');
```

CREATE TYPE ...
- erstellt einen zusammengesetzten Datentyp
- Für Datenschema und Stored Procedures
- Auch für ENUM Type verwendet
- Keine Constraints-Angaben

```
create type easteregg as (
   outer: text,
   inner: text,
);
create type traffic_light_t as
   enum('red', 'yellow', 'green');
```

One is a completely new type, the other is just a type that is
made up of already known types.
Essentially the right one is a class or struct.
The left is a completely new thing, that is not yet implemented.

Optimization and indexing

The basics of indexing is that is saves
time on queries, but it uses more space,
and needs to be redone on udpate/insert.

Data stored in Pages / Heap (Collection of Pages)
Indexing on either can be possible.
ex: Page index -> Primary key index
ex: Heap Index -> index on tables
PSQL does table cluster indexing
instead of integrated indexing -> key value (both indexed)

B-Tree: the default, can index multiple entries (only btree!)
CREATE UNIQUE INDEX name ON table (column [1,2...])

Hash Index: just like hashmap in programming
good for single or small multiple queries
bad for entire tables etc -> collision
hashing might take a long time with a lot of data.
B-tree almost always better!

GiST: balanced/treelike , Range/neighbor/fulltext search
used for geometric datatypes. SP-Gist for unbalanced trees.

GIN: General Inverted Index "list of words
that point to documents" wtf?. Good for duplicates.
Good for hstore,Json,Arrays
Vergleich zu GiST:
- Ca. 3x schnellerer Zugriff
- Ca. 2 – 3x mehr Diskplatz
- Ca. 2 – 3x länger bis Index erstellt ist
- Ca. 10x langsamer bei Update

Bitmap indexing: Bitmap -> 0 1 stores Booleans/Enums
very fast read / slow update
in postgres **only implicit** use Brin instead.

BRIN: Block Range Index, stores min/max values as blocks
good for range search, sorted data, small disk usage
data is naturally sorted, address next to postal code.

Bloom Index -> equality search,
Trigram Index -> Full text search
RUM , non-default-GIN jsonb-path-ops

creating index:
CREATE INDEX <indexname> ON <table(attribute)>;
and: DROP INDEX <indexname>;

default index order for psql is btree,ASC,NULL first

Index-Variationen
- Zusammengesetzter Index ->query on phone return phone and name
  - Bsp.: CREATE INDEX idx_addr ON addr(phone,name); kann für Queries auf phone und "phone AND name" genutzt werden, sowie für bestimmte Q. auf name; jedoch nicht für Suffix-Q. "... LIKE %name;"
- Index mit INCLUDE similar
  - Bsp.: CREATE INDEX idx_addr ON addr(phone) INCLUDE (name) ;

Index-Variationen ff.
- Partieller Index index if(condition)
  - Bsp.: CREATE INDEX idx_addr ON addr(status) WHERE status='active';

index on function()
- Indexe mit Funktionen / Ausdrücke ("Funktionaler I.")
  - Bsp.: CREATE INDEX idx_addr ON addr(lower(name));
- Nicht nur PostgreSQL!

PG planner join strategies: Nested Loop,Merge,Hash
-Nested Loop: for r in right row r == for l in left row....
good for small tables, easy to setup
-Merge: Merge rows one after the other
higher starting cost, good for bigger tables
-Hash: Hash the row then compare to other row
equality check only , high starting cost, low execution cost

PG planner scans: Full,Index,Index Only, Bitmap
-full scans the entire table
-index scans index and more (if necessary)
-index only only scans index
-Bitmap scans the bitmap generated by an index.

The steps of optimization

1. generate the plan of transaction
2. reform the term to optimize performance
without knowledge of the internal structure.
-> all values are considered equal
3. optimization based on: available indexes, analysis & costs
4. generate all possible plans to calculate cost
5. analyze said plans -> how many tuples, what kind...
6. profit?

selectivety this is the ratio of tuples a query returns
low selectivety would mean high number of rows
an example is select * from table where sex='Male'
the opposite would be high selectivety.

density this is the ratio of duplicates a query returns
the more duplicates the higher the density.
you can therefore also make graphs about the distribution
of density -> names a-c low g-l high

best practices: index only when selectivety over 10%
numeric comparison over text, join over subquery,
use short attributes, understand the query -> user,
don't select * from, don't use cross products.

distributed database systems

the advantages and disadvantages are obvious:
+ better performance – more complex
+better reliability + better management

homogeneous database system:
> all nodes have identical software
> all nodes know about each other and work together
> shows itself to the user as one big system

heterogeneous database system:
> nodes can have different software
> as well as different schemas -> can lead to problems
> nodes might NOT know about each other.

heterogeneous systems can be federated or unfederated
> unfederated -> no local users
> federated -> either tightly coupled with global schema
or loosely coupled using export schema

fragmentation: this is the splitting of schemas
into multiple Nodes -> table 1 in node1
table 2 in node 2.

In Psql the horizontal fragmentation happens in 3 ways:
> 1 or more attributes for "partitioning key"
> "list" explicit designation
> hash function (ex: Modulo)

In Graph stores this would be called "sharding"
an example for this is the MongoDB
horizontal partitioning and allocation within a single node

replication: this is the duplication of data in schemas
this means table 1 might be on node1 and 2.
vertical -¿ splitting of columns
row1 in node1, row2 in node2
horizontal -¿ splitting of rows
part of column in node1 part of column in node2

unidirectional: Single-Master

bidirectional: Multi-Master

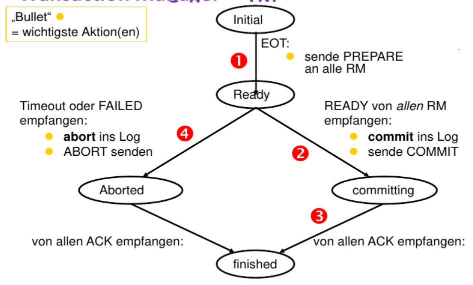
synchronous OR asynchronous

allocation: this is the distribution of work to the nodes
node1 might handle query or part of query1
while node2 handles something else.

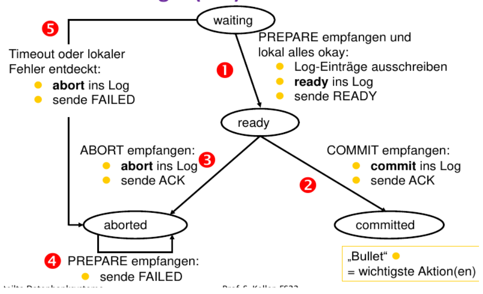
as already stated above, the user doesn't see anything about
fragmentation or similar, the user simply interacts
with the schema and executes transactions.
these **transactions are always local**
This means that the user will talk to **Transaction Manager**
that will handle the transaction and call the necessary
functions on the **Resource Managers** (inside nodes).

Two Phase Commit Protocol

Transaction Manager TM



Resource Manager (RM)



case TM failed/restart:
> if the TM crashes before the commit message -> abort
> if the TM crashes after RM respond ready -> block RM
this is one of the main problems btw...

case RM failed/restart:
> if no entry in log, RM aborts
> if READY-Entry available -> RM asks TM what to do.
> if COMMIT-Entry available -> RM redoes transaction

case message dropped:
> if the prepare statement gets lost,
or the RM doesn't respond, then the TM
simply aborts the transaction for all.
> if RM doesn't get a response in READY state
then the RM will remind the TM until it gets one.

There are different 2PC protocols, PSQL, MYSQL, X/Open,
Java Transaction API, Orace Transaction Manager, Microtroll
+ often used / proven system + guarantees ACID
– slow – blocks transactions often
!!only use this when the complexity calls for it!!

Query handling:
there are 3 ways of handling a queries distributed databases:
> Ship Whole -> run query locally and remove duplicates
> Push Down -> split query when needed
> Fetch-as-needed -> Send join attributes to the correct node

NoSQL: Key/Value Stores

Why even NoSQL? -> fast, lots of data, webbased, scalable
RDBMS -> relational has the problem of complexity
it doesn't integrate seamlessly into programming languages
it therefore doesn't make sense to use this as
something like a webstore. -> !!! OR MAPPING !!!

attributes of NoSQL:
> easy to use API (http) > made for big volumes of data
> provided as cloud storage > not relational, schema free
> BASE instead of ACID > no query norm (other than SQL)

The Cap Theorem
The CAP theorem dicates, that you can only have
2 of 3 desired traits of a database, such as
consistency, parition tolerance and availability.
traditional databases have the **ACID** philosophy which is both
consistent and tolerant, however it often blocks transactions.
here comes **NoSQL with the BASE** philosophy
it is both **available** at all times and **tolerant**,
however it is **NOT consistent during transactions**.
Only after those have stopped will the system get consistent.
>tolerance means the system will work despite partial outage<

BASE THEOREM
>> Basically Available:
The database will not block transactions
any and all requests will be responded to (can still fail though!)
>> Soft state:
The state of the system can change even without input,
therefore we consider this to be a soft (not fixed) state
>> Eventual consistency:
The system will become consistent after end of transactions.

Key-Value Database

- A simple hash table accessible only through its primary key
- Basically a table with two columns: ID and VALUE
- The value is a blob that the data store just stores: it can be text, JSON, XML, and anything else.
- Operations:
  - get the value for the key
  - put a value for a key (if the key already exists the corresponding value is overwritten)
  - delete a key from the data store

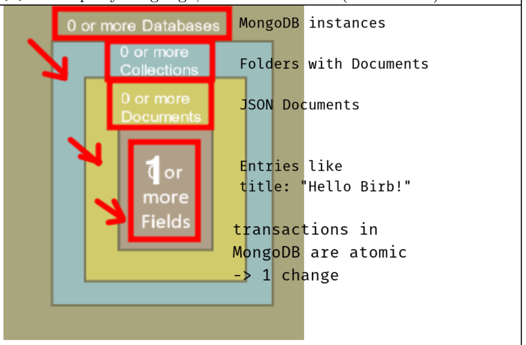
notable examples: Riak, Redis, Memcached, Berkeley DB
HamsterDB , Amazon DynamoDB, Project Voldemort

Key/Value Stores are usually used for this:
>> Storing Session Information
>> User Profiles, Preferences, Configs
>> Shopping carts (LMAO)
They are **NOT** to be used in these cases:
<< relationship among data , multioperation transactions
<< query by data, operation by sets

NoSQL Aggregation database
> Arrays (Array Store)
> Dictionaries (Key/Value Stores)
> nested Structures(Document-databases)

NoSQL Document Store
>> Maps a key to a structured Document
>> flexible schema , Document stored in **JSON** or BSON
>> Examples: **MongoDB**, CouchDB
term comparison to RDBMS
instance -¿ instance/database, table -¿ collection
row -¿ Document , rowid -¿ _id/objectId

MongoDB
>> JSON, partitioning via sharding, FOSS :)
>> own query language, Document Store (no Schema)



```
var class = {
  _id: ObjectId ("509980df3"),
  course: {code: "Dbs2",
           title: "Advanced DB"},
  year: 2016,
  students: ["Peter", "Manuel",...],
  no_of_st: 34
}
```

- The document contains values of varying types
  - The primary key is _id and it is of the ObjectId type,
  - The field course is a subdocument, and
  - students is an array of strings

The documents are actually stored as BSON
the binary version of JSON with a bit more data.
Also there are certain restriction on naming...

_id

- The field _id is reserved for use as a document's primary key
  - Its value must be unique in the collection,
  - It is immutable,
  - May be of any type other than an array or a regular expression type
  - MongoDB creates a unique index on the _id field during the creation of a collection.
  - It is always the first field in the documents
- The following are common options for storing values for _id:
  - Use an ObjectId,
  - Use a natural unique identifier, if available
    - This saves space and avoids an additional index,
  - Generate an auto-incrementing number
- **ObjectId** is a 12-byte BSON type, constructed using:
  - A 4-byte value representing the seconds since the Unix epoch,
  - A 3-byte machine identifier,
  - A 2-byte process id, and
  - A 3-byte counter, starting with a random value
- ObjectIds are small, most likely unique, and fast to generate
- MongoDB uses ObjectIds as the default value for the _id field if the _id field is not specified by a client
- Additional benefits of using ObjectIds for the _id field:
  - In the mongo shell, you can access the creation time of the ObjectId, using the getTimestamp() method,
  - Sorting on ObjectId values is roughly equivalent to sorting by creation time.

MongoDB Queries

- Expressed via JSON with simple constructs

```
// in orders
{ "orderId":99,
   "customerId":"883c2c5b4e5b",
   "orderDate":"2014-04-23",
   "orderItems":[
    { "product": {"id":27, "name":"NoSQL Distilled"}, "price": 32.45 }
    { "product": {"id":55, "name":"Java 4 all"}, "price": 41.33 }
   ], }
db.orders.find()
db.orders.find({"customerId":"883c2c5b4e5b"})
db.order.find({customerId:"883c2c5b4e5b"},
{orderId:1,orderDate:1}) db.orders.find({"orderItems.product.name":/NoSQL/})
```

sharding

horizontal fragmentation into nodes
reduces IO due to outsourcing to other servers

The shard key -¿ fragmentation key should be:
>> distributed equally in the data
>> is a field/function or a combination of fields
should be not unique, but can be a hash on PK?....
default is hash on _id aka PK
shards can usually not be changed -> redo database.

MongoDB only offers few constraints like
the unique index constraint.

advantages:
>> simple query language, >> no schema
disadvantages:
>> no constraints
>> no real joins (workaround with lookup)
>> slow/questionable transactions -> BASE
>> security -> default open in the web

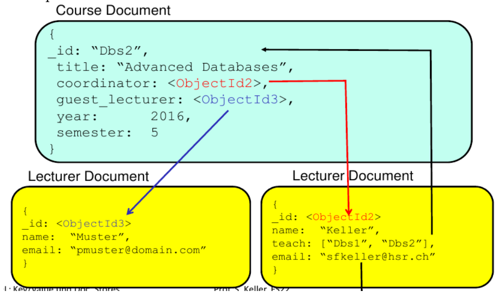
MongoDB Transactions
updates work similar to increasing an array
it even automatically pushes it to a new document
if the current one is too big!
however you can avoid relocation by using
referencing instead of embedding.

Performance hindering features:
>> Atomicity of writes — >> Document Growth
>> Sharding — >> Indexes and Capped Collections

MongoDB works with **Replica Sets**
child node has the data as well
in case of outage you can still get your data
due to the nature of MongoDB this happens
without having a specific distributed database.
Also the nodes will pick a new Master if the
old one goes down.

embedded: ALL IN ONE FILE
Preferred for >> ONE-TO-ONE
>> ONE-TO-MANY without many overlaps
Warning with this method, uncontrolled growth of document

referenced: Stored in multiple files
Preferred when dealing with complex relationships
this is the advantage Document stores have over key/value!
example of reference:



```
{
_id: "Dbs2",
 title: "Advanced Databases",
 coordinator: <ObjectId2>,
 guest_lecturer: <ObjectId3>,
 year:       2016,
 semester:  5
}
```

```
{
_id: <ObjectId3>
name:  "Muster",
email: "pmuster@domain.com"
}
```

```
{
_id: <ObjectId2>
name:  "Keller",
teach: ["Dbs1", "Dbs2"],
email: "sfkeller@hsr.ch"
}
```

- MongoDB use one of two methods for relating documents:
  - Manual references where you save the _id field of one document in another document as a reference
    - These references are simple and sufficient for most use cases
  - The other method is to use DBRefs
- MongoDB documentation recommends using manual references

and here how to create it:

```
use mydb
var coordinator_id = ObjectId()
var guest_lec_id = ObjectId()

db.class_ref.insert({
 _id: "Dbs2",
 title: "Advanced Databases",
 coordinator: coordinator_id,
 guest_lecturer: guest_lec_id,
 year:      2016,
 semester:  5
})
```

in general, only use references for things that make sense,
for example don't make a reference for a blog post
or for a persons gender etc.
embed the small stuff, reference the big stuff

- MongoDB automatically creates a unique index on the _id field
- Indexes on fields (other than _id) that appear often in queries improve performance for common queries
- Indexes are built as BTrees (facilitating range queries)

createIndex() or ensureIndex()

- db.collection.createIndex(keys, options)
- Parameters:
  - keys of the type document:
    - For each field to index, a key-value pair with the field and the index order: 1 for ascending or −1 for descending
  - options of the type document (optional)
- The most important options:
  - unique of the type Boolean:
    - The default value is false
  - name of the type string:
    - If unspecified, MongoDB generates an index name

```
db.collection.ensureIndex(
{_id: 1, year: -1}, {unique: true}
)
```

Capped Collection (*)

- Capped collections are fixed-size collections that support those high-throughput operations that insert and retrieve documents based on insertion order
- Capped collections work in a way similar to circular buffers: once a collection fills its allocated space, it makes room for new documents by overwriting the oldest documents in the collection

MongoDB CRUD

MongoDB Read Operation

- The read operation is defined from within the mongo shell
- It uses `db.collection.find()` method
- The method accepts: selection criteria, projection list, and modifiers as its arguments
- Selection ctiteria:
  - Comparison,
  - Existence,
  - Logical junctions (and, or,...),
  - Regular expressions,
  - Array selection objects
- Projection list contains either a:
  - List of fields to return, or
  - List of fields not to return,
  - The only exception is _id field that may be marked as not to return among the list of fields to return
- The read operation returns a cursor to matching documents
  - In mongo shell, up to 20 first documents are displayed on the standard output
- Cursor can be used to write handy scripts

MongoDB Aggregation

- Three ways to do aggregation:
  - Pipeline (preferred),
  - Map Reduce, and
  - Single purpose
- Aggregation Pipeline consists of stages
- Stage operators:
  - `$match, $group, $project, $sort, $skip`,...
- Each stage operator contains expression(s) that specify transformations of documents
- Expression operators:
  - Boolean, Arithmetic, Comparison, String, Set, Date, Accumulator,...
- Expressions in the `$group` and `$project` stages use field path to access fields in the input documents
- Examples show use cases

Lookup mit Aggregation (= aka Joins!)

- Lookup mit Aggregation (= aka Joins!) ab Version 3.2!

```
db.absences.aggregate([ {
   $lookup: {
      from: "holidays",
      pipeline: [
         { $match: { year: 2018 } },
         { $project: { _id: 0, date: { name: "$name", date: "$date" } } },
         { $replaceRoot: { newRoot: "$date" } } ],
      as: "holidays"
   }
} ])
```

- Entspricht Left Join mittels Subquery:

```
SELECT *, holidays
FROM absences
WHERE holidays IN (SELECT name, date FROM holidays
WHERE year = 2018);
```

- Siehe MongDB Docs $lookup

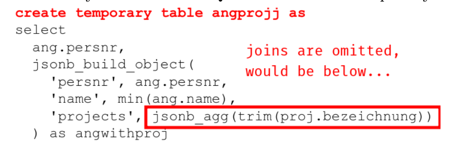
MongoDB Write Operation

- There are three classes of write operations in MongoDB:
  - Insert that adds a new document to a collection,
  - Update that modifies an existing document, and
  - Remove that deletes an existing document from a collection.
  - The update and remove operations allow specifying criteria or conditions that identify documents (to be modified, or removed)
    - The syntax of the criteria is the same as in find() method
- For all inserts and updates, MongoDB modifies each document in isolation
  - Clients never see documents in an intermediate state
- For multi-document operations, MongoDB does not provide any multi-document transactions or isolation
- The method `db.collection.insert()` writes a document into a collection
  - If there is no `_id` field specified in the document, MongoDB generates it
- The method `db.collection.update()` modifies an existing document, or may upsert a new one
- The `update()` method can also rename or delete a field
- The methods `db.collection.remove()` and `drop()` are used to delete `just` documents or both documents and indices of a collection, respectively

NoSQL Graph Stores

made for compact objects with typed aggregations
sometimes without schema -> implicit schema
or not uniform data

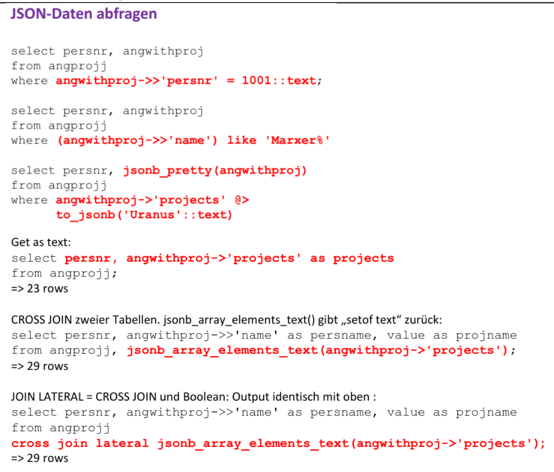
easiest way to convert PSQL to JSON is a temporary table.



```
 create temporary table angprojj as
 select
   ang.persnr,                joins are omitted,
   jsonb_build_object(        would be below...
     'persnr', ang.persnr,
     'name', min(ang.name),
     'projects', jsonb_agg(trim(proj.bezeichnung))
   ) as angwithproj
```

you can also directly **select json_build_object**
or **select json_build_object(jsonb_agg(tmp))**
also note the jsonb_agg(trim (proj.bezeichnung))
for simplicity, the joins on the picture are removed.

The jsonb agg is necessary for aggregation
otherwise the join wouldn't work for JSON.



JSON-Daten abfragen

```
select persnr, angwithproj
from angprojj
where angwithproj->>'persnr' = 1001::text;

select persnr, angwithproj
from angprojj
where (angwithproj->>'name') like 'Marxer%'

select persnr, jsonb_pretty(angwithproj)
from angprojj
where angwithproj->'projects' @>
      to_jsonb('Uranus'::text)
```

Get as text:
```
select persnr, angwithproj->'projects' as projects
from angprojj;
```
=> 23 rows

CROSS JOIN zweier Tabellen. jsonb_array_elements_text() gibt „setof text“ zurück:
```
select persnr, angwithproj->>'name' as persname, value as projname
from angprojj, jsonb_array_elements_text(angwithproj->'projects');
```
=> 29 rows

JOIN LATERAL = CROSS JOIN und Boolean: Output identisch mit oben :
```
select persnr, angwithproj->>'name' as persname, value as projname
from angprojj
cross join lateral jsonb_array_elements_text(angwithproj->'projects');
```
=> 29 rows

JSON — SQL