Index

Doel: beïnvloeden van de verwerkingstijd!
Rijen worden in bestanden opgeslagen. Een bestand bestaat uit pagina's. Als een rij opgehaald wordt : (1) de betreffende pagina wordt opgehaald (2) de betreffende rij wordt opgehaald

Werking

2 methodes voor het opzoeken:
● Sequentiële zoekmethode: rij voor rij → Tijdrovend en inefficiënt
● Geïndexeerde zoekmethode: index (B-tree) → Boom, Knooppunten, Leafpage
→2 methodes: Zoeken van rijen met een bepaalde waarde
Doorlopen van de hele tabel via een gesorteerde kolom (geclusterde index)
Bij aanpassing in tabel: index aangepast. Index: ook op niet-unieke kolom. Opgelet: opslagruimte!

Create index

Geen ANSI of ISO specificatie. Bij postgresql:

●`CREATE INDEX spelers_postcode_idx ON spelers (postcode asc);` ●`CREATE UNIQUE INDEX spelers_naam_vl_idx ON spelers (naam, voorletters); -- UNIQUE !`	●`CREATE INDEX spelers_naam_vl_partial_idx ON spelers (naam, voorletters) WHERE spelersnr < 100;` ●`CLUSTER spelers USING speler_naam_vl_idx;`
●`REINDEX INDEX een_index;` ●`REINDEX TABLE een_tabel;` ●`REINDEX DATABASE een_database;`	●`ALTER INDEX groot_idx SET TABLESPACE ergens_anders;` ●`DROP`

(+) index versnelt verwerking (–) opslagruimte; elke mutatie vraagt aanpassing → verwerking vertraagt

Richtlijnen voor keuze van kolommen

Unieke index op kandidaatsleutels ● Index op refererende sleutels ● Index op kolommen waarop geselecteerd wordt–Grootte van de tabel–Kardinaliteit (verschillende waarden) van de tabel–Distributie van de waarden
● Index op een combinatie van kolommen● Index op kolommen waarop gesorteerd wordt

Speciale indexvormen

●**Multi-tabelindex** → op kolommen in meerdere tabellen ●**Virtuele-kolomindex** → op een expressie ●**Selectieve index** → op een gedeelte van de rijen	●**Hash-index** → op bass van adres op pagina ●**Bitmapindex** → interessant als er veel dubbele waardes zijn

Indexvormen

●B-tree → goeie standaard, = <>
●GIN(Generalized Inverted Index) → efficiënt bij dubbels, meerdere opzoekwaarden per veld; alternatief voor B-tree
●GiST(Generalized Search Tree) →voor clusters volgens afstandsmaat. Vergelijken van intervallen bv. Veel mogelijkheden, minder performant
●Sp-GiST(space-partitioned GiST) → niet overlappende GiST
●BRIN: bevat, grote geclusterde data, kleine index maar ook niet heel sterk

Optimaliseren

1. Vermijd de OR-operator → index wordt meestal niet gebruikt
   - Vervang door conditie met IN of door 2 selects met UNION
2. Onnodig gebruik van UNION → zelfde tabel meerdere malen doorlopen. Herformuleer naar 1 select
3. Vermijd NOT-operator → index niet gebruikt. Vervang door vergelijking ofzo

4. Isoleer kolommen in condities → kolom in berekening of scalaire functie betekent geen gebruik van index. Bv. Where jaartoe + 10 = 1990 → where jaartoe = 1980
5. Gebruik de BETWEEN operator. And gebruikt meestal index niet.
   - Where jaartoe >= 1985 and jaartoe <= 1990 → where jaartoe between 1985 and 1990
6. Pas op met LIKE → index wordt niet gebruikt als patroon begint met % of _
   - Geen alternatief, tenzij: HHHHHHEJLHELEJH
7. Redundante condities bij JOIN : om SQL te verplichten om een bepaald pad te kiezen
   - where boetes.spelersnr = spelers.spelersnr and boetes.spelersnr = 44
     → where boetes.spelersnr = spelers.pelersnr and boetes.spelersnr = 44 and spelers.spelersnr = 44
8. Vermijd de HAVING-component → index niet gebruikt. Zoveel mogelijk in WHERE dus.
9. SELECT-component zo compact mogelijk → onnodige kolommen weg. Bij gecorreleerde subquery met exists : één expressie bestaande uit één constante.
   - select spelersnr, naam from spelers where exists (select '1' from boetes where boetes.spelersnr = spelers.spelersnr)
10. Vermijd DISTINCT : verwerkingstijd verlengd. Vermijden wanneer overbodig.
11. ALL-optie bij set operatoren (union all). Zonder ALL → verwerkingstijd verlengd. Data moeten gesorteerd worden om dubbels eruit te halen
12. Kies outer-joins boven UNION. Union verlengt verwerkingstijd.
13. Vermijd datatype-conversies → verlengt verwerkingstijd.
14. Grootste tabel als laatste
15. Vermijd ANY- en ALL-operatoren → index niet gebruikt. Vervang door min of max

Check met Explain en Explain Analyze.

VIEWS

tabel die zichtbaar is voor de gebruiker maar geen opslagruimte inneemt. De inhoud van een view wordt afgeleid bij het gebruik van een SELECT view. Een view kan dus maar opgebouwd worden op basis van gegevens die in andere tabellen opgeslagen zitten.

```
CREATE VIEW leeftijden (spelersnr,
leeftijd) AS SELECT spelersnr,
2008 – year(geb_datum)
FROM spelers;
```

```
CREATE VIEW leeftijden AS
SELECT spelersnr,
2008 – year(geb_datum) AS leeftijd
FROM spelers;
```

Vereenvoudigt routinematige instructies. Reorganiseren van tabellen. Stapsgewijs opzetten van SELECT-instructies. Beveiligen van gegevens. View bevat geen rijen.
Expliciete definitie is verplicht als kolom bestaat uit een functie of berekening:

```
CREATE VIEW leeftijden (spelersnr,
leeftijd) AS SELECT spelersnr,
2008 – year(geb_datum)
FROM spelers;
```

```
CREATE VIEW leeftijden AS
SELECT spelersnr,
2008 – year(geb_datum) AS leeftijd
FROM spelers;
```

WITH CHECK OPTION (achteraan) controleert :update : aangepaste rijen behoren nog tot view
- insert : nieuwe rij behoort tot view - delete : verwijderde rij behoort tot view

Beperkingen bij muteren

SELECT, INSERT, UPDATE, DELETE van views. Maar mutatie mag alleen als :

- ●View moet direct/indirect gebaseerd zijn op één of meerdere basistabellen
- ●Select mag geen distinct bevatten
- ●Select mag geen aggregatiefunctie bevatten
- ●From mag slechts één tabel bevatten
- ●Select mag geen GROUP BY bevatten
- ●Select mag geen ORDER BY bevatten
- ●Select mag geen set-operatoren bevatten
- (update) Virtuele kolom mag niet gewijzigd worden
- (insert) In Select moeten alle not null-kolommen staan

Beveiliging

SQL gebruiker : moet gekend zijn; Wachtwoord; Expliciete toekenning van bevoegdheden

- ●CREATE USER : creëert een user Vb. Create user Frank identified by Frank_pw
- ●ALTER USER : verandert het paswoord Vb. Alter user Frank identified by Frank_pasw
- ●DROP USER : verwijdert een user Vb. Drop user Frank

Vensterfuncties

these functions must be invoked using window function syntax; that is an OVER clause is required.

Function	Return Type	Description
`row_number()`	`bigint`	number of the current row within its partition, counting from 1
`rank()`	`bigint`	rank of the current row with gaps; same as `row_number` of its first peer
`dense_rank()`	`bigint`	rank of the current row without gaps; this function counts peer groups
`percent_rank()`	`double precision`	relative rank of the current row: (`rank` - 1) / (total rows - 1)
`cume_dist()`	`double precision`	relative rank of the current row: (number of rows preceding or peer with current row) / (total rows)
`ntile(num_buckets integer)`	`integer`	integer ranging from 1 to the argument value, dividing the partition as equally as possible
`lag(value anyelement [, offset integer [, default anyelement ]])`	`same type as value`	returns `value` evaluated at the row that is `offset` rows before the current row within the partition; if there is no such row, instead return `default` (which must be of the same type as `value`). Both `offset` and `default` are evaluated with respect to the current row. If omitted, `offset` defaults to 1 and `default` to null
`lead(value anyelement [, offset integer [, default anyelement ]])`	`same type as value`	returns `value` evaluated at the row that is `offset` rows after the current row within the partition; if there is no such row, instead return `default` (which must be of the same type as `value`). Both `offset` and `default` are evaluated with respect to the current row. If omitted, `offset` defaults to 1 and `default` to null
`first_value(value any)`	`same type as value`	returns `value` evaluated at the row that is the first row of the window frame
`last_value(value any)`	`same type as value`	returns `value` evaluated at the row that is the last row of the window frame
`nth_value(value any, nth integer)`	`same type as value`	returns `value` evaluated at the row that is the `nth` row of the window frame (counting from 1); null if no such row

last_name	salary	department
Jones	45000	Accounting
Adams	50000	Sales
Johnson	40000	Marketing
Williams	37000	Accounting
Smith	55000	Sales

Lets assume that you wanted to find the highest paid person in each department. There's a chance you could do this by creating a complicated stored procedure, or maybe even some very complex SQL. Most developers would even opt for pulling the data back into their preferred language and then looping over results. With window functions this gets much easier.

```
SELECT last_name,
    salary,
    department,
    rank() OVER (
     PARTITION BY department
     ORDER BY salary
     DESC
    )
FROM employees;
```

← First we can rank each individual over a certain grouping:

last_name	salary	department	rank
Jones	45000	Accounting	1
Williams	37000	Accounting	2
Smith	55000	Sales	1
Adams	50000	Sales	2
Johnson	40000	Marketing	1

```
SELECT *
FROM (
    SELECT
           last_name,
           salary,
           department,
           rank() OVER (
             PARTITION BY department
             ORDER BY salary
             DESC
            )
    FROM employees) sub_query
WHERE rank = 1;
```

← Hopefully its clear from here how we can filter and find only the top paid employee in each department:

last_name	salary	department	rank
Jones	45000	Accounting	1
Smith	55000	Sales	1
Johnson	40000	Marketing	1

The best part of this is Postgres will optimize the query for you versus parsing over the entire result set if you were to do this your self in plpgsql or in your applications code.

Common Table Expressions

Mogelijk nut: ●Tijdelijke tabel ~ view. ●Beetje zoals subqueries, maar losser gelinkt aan de main query. ●Recursieve tabellen recursief ontleden	Sleutelwoord: WITH Gebruik met SELECT, DELETE, UPDATE, INSERT main queries Kan op zichzelf een SELECT, DELETE, UPDATE of INSERT auxiliary table zijn

__CTE as temporary table

Global Structure:	**Structure of *def temp table* :**
WITH [def temp table 1][, def temp table 2] [, def temp table 3].... Main query that can make use of earlier defined temp tables	Temp_table_name AS (query)

Voorbeeld:

update fines by 10% of all competition players that have never been in the board

```
WITH comp_spelers AS (
        SELECT spelersnr, naam, voornaam
        FROM spelers            WHERE bondsnr IS NOT NULL
), non_bestuur AS (
        SELECT *
        FROM bestuursleden b RIGHT OUTER JOIN spelers s
        ON (b.spelersnr = s.spelersnr)
        WHERE b.spelersnr IS NULL
)
UPDATE boetes           SET bedrag = bedrag * 1.1
FROM comp_spelers c INNER JOIN non_bestuur n ON (c.spelersnr = n.spelersnr)
WHERE  c.spelersnr = b.spelersnr
```

__Recursive CTE

Structure:

```
WITH RECURSIVE aux_table (attr1, attr2, ...) AS (
        initialization / getting base data
UNION ALL
        building the output by recursive parsing
)
SELECT
FROM aux_table;
```

```
CREATE TABLE kings (
        id INTEGER NOT NULL,
        naam VARCHAR(30),
        zoon VARCHAR(30),...
)
INSERT INTO kings (id, naam) VALUES
(1, 'Louis I', 'Louis II'), (2, 'Louis
 II', 'Louis III'), ..., (14, 'Louis XIV', 'Louis XV')
```

Give Louis V and his 5 successors

```
WITH RECURSIVE lijst (naam, zoon) AS (
        SELECT naam, zoon
        FROM kings
        WHERE naam = 'Louis V'

UNION ALL
        SELECT K.naam, K.zoon
        FROM lijst L INNER JOIN kings K
                ON (K.naam = L.zoon)
)
SELECT naam
FROM lijst;
```

→ Init stage generates **first** record for table lijst in line with the definition, being attributes naam, zoon i.e.: 'Louis V', 'Louis VI'

→ In build up stage, we continue from the first generated row in accordance with the recursion defined (K.naam = L.zoon). As such the second, third, ... records are generated based on the previous record:
This leads to the following rows:
'Louis VI', 'Louis VII' 'Louis VII', 'Louis VIII'

→ From the generated table lijst, we select to output only the *naam* attribute

Needs to stop at five, though. Either stop criterium or limit number of recursions

Give Louis V and his successors till Louis X

```
WITH RECURSIVE lijst (naam, zoon) AS (
        SELECT naam, zoon
        FROM kings
        WHERE naam = 'Louis V'
UNION ALL
        SELECT K.naam, K.zoon
        FROM lijst L INNER JOIN kings K
                ON (K.naam = L.zoon)
        WHERE K.naam = 'Louis X'
)
SELECT naam
FROM lijst;
```

Recursion depth control

```
WITH RECURSIVE lijst (naam, zoon, nr) AS (
        SELECT naam, zoon, 0
        FROM kings
        WHERE naam = 'Louis V'
UNION ALL
        SELECT K.naam, K.zoon, nr + 1
        FROM lijst L INNER JOIN kings K
                ON (K.naam = L.zoon)
        WHERE nr < 5
)
SELECT naam
FROM lijst;
```

ODMS

Het Objectgeoriënteerde database model is een databasemodel waarin net zoals in objectgeoriënteerde programmeertalen met objecten wordt gewerkt. Het doel van zo'n database is het invoegen van dergelijke objecten in de database zo eenvoudig mogelijk te maken. Zo wenst men de objecten die men gebruikt in een objectgeoriënteerde taal direct te kunnen opslaan in de database, zonder tupels zoals in een relationele database te moeten gebruiken.

De meeste Object Database management systemen ODBMS ondersteunen een querytaal en maken daarmee een declaratieve aanpak mogelijk. Hoe dit wordt aangepakt verschilt van product tot product. Er is reeds een poging gedaan om dit te standaardiseren in Object Query Language OQL.
Men kan data sneller opvragen doordat er geen join operaties nodig zijn, men kan de pointers rechtstreeks volgen. Dit is dan een voordeel ten opzichte van relationele databases. Veel ODBMS laten het ook toe om verschillende versies van objecten bij te houden.

Object Definition Language (ODL) is the specification language defining the interface to object types conforming to the Object Model. This language's purpose is to define the structure of an Entity-relationship diagram.

ODL-examples

```
class STUDENT
(extent PERSISTENT_STUDENTS /*persistent*/
Key Ssid)
{attribute string Ssid;
attribute string FamilieNaam;
attribute ..
relationship REEKS zitIn
inverse REEKS::heeftStudenten;
void verplaatsStudent(in string NewReeks)
raises(NewReeksBestaatNiet)
}
```

```
class REEKS
(extent REEKSEN
Key Rnaam)
{attributestring Rnaam;
attribute ..
relationship set<REEKS> heeftStudenten
inverse REEKS::zitIn;
void voegStudentToe(in string NewReeks)
raises(NewReeksBestaatNiet)
}
```

ODMS : kolom objecten

```
class AUTO
(..)
{attribute string Snrplt;
attribute STUDENT Eigenaar;
attribute ..}
```

ODMS : geneste objecten

```
class STUDENT
(extent STUDENTEN
.. )
{..
attribute struct Adres{string straat; string huisnr;..}
```

Collections

- set<type>
- bag<type>
- list<type>
- array<type>
- dictionary<key,value>

Overerving

```
class BRAVE_STUDENT extends
STUDENT
(..)
{..
attribute string
nieuwJaarBrief;
..}
```

OQL voorbeelden

```
select S.FamilieNaam
from S in PERSISTENT_STUDENTS
where S.Ssid = '12345';

REEKSEN;
STUDENT1.Adres;

select distinct S.Ssid
from S in REEKS1.heeftStudenten;
```

ORDBMS

An object-relational database management system (ORDBMS), is a database management system (DBMS) similar to a relational database, but with an object-oriented database model: objects, classes and inheritance are directly supported in database schemas and in the query language. In addition, just as with pure relational systems, it supports extension of the data model with custom data-types and methods.

An object-relational database can be said to provide a middle ground between relational databases and object-oriented databases (object database). In object-relational databases, the approach is essentially that of relational databases: the data resides in the database and is manipulated collectively with queries in a query language; at the other extreme are OODBMSes in which the database is essentially a persistent object store for software written in an object-oriented programming language, with a programming API for storing and retrieving objects, and little or no specific support for querying.

Create type spelersnr as integer;
Create table spelers
(spelersnr spelersnr not null,...);
Drop type spelersnr;

type gebruiken, → machtiging

grant usage on type geldbedrag to Jim
revoke usage on type geldbedrag to Frank

select * from boetes where bedrag > geldbedrag(50)
select * from boetes where decimal(bedrag) > 50
insert into boetes (betalingsnr, spelersnr, datum, bedrag)
values (betalingsnr(12), spelersnr(6), '1980-12-08',
geldbedrag(100.00)) *Bedrag is van type geldbedrag*

Definiëren van operatoren in de vorm van scalaire functies
bedrag1 + bedrag2 = ongeldig
→decimal(bedrag1) + decimal(bedrag2)
→create function « + » (geldbedrag, geldbedrag) returns geldbedrag
source « + » (decimal(), decimal())

Named row-datatype: groeperen van waarden die logisch bij elkaar horen
create type adres as
(straat char(15) not null, huisnr char(4),
postcode char(6), plaats char(10)not null) ;

Unnamed row → zonder naam te geven

create table spelers (spelersnr smallint primary key,
… woonadres row (straat char(15) not null, huisnr char(4), postcode char(6), plaats char(10) not null),
telefoon char(10) , ...

Opaque-datatype → niet afhankelijk is van een basisdatatype. Definiëren van functies om hiermee te werken is noodzakelijk. Vb. create type tweedim(internallength = 4) ;

Getypeerde tabel
Eenvoudig gelijkende tabellen maken

create type t_spelers as(spelersnr integer not null, naam char(15) not null, ...
bondsnr char(4));
create table spelers of t_spelers
(primary key spelersnr) ;

Integriteitsregels op datatypes
Beperkingen op de toegestane waardes

create type aantal_sets as smallint
check (value in (0, 1, 2, 3));

create table wedstrijden(wedstrijdnr integerprimary key, teamnr, …, verloren aantal_sets not null) ;

ALTER TYPE : wijzigen van datatype

Vb.
alter type aantal_sets as smallint
check (value between 0 and 4)

Als conditie strikter is, wijziging geweigerd tot al deze waarden in de tabel aangepast zijn

Sleutels en indexen → Is volledig analoog bij zelfgedefinieerde datatypes
Bij named row-datatypes : ●op de volledige waarde ●op een deel ervan

create type adres as
(straat char(15) not null,
huisnr char(4),
poscode char(6),
plaats char(10) not null);

create type buitenlands_adres as
(land char(20) not null) under adres ;

Overerving van datatypes → Alle eigenschappen van één datatype worden overgeërfd door een ander (supertype en subtype)

create table spelers
(spelersnr smallint primary key, ...
woonadres adres,
vakantieadres buitenlands_adres , ...
);

Koppelen van tabellen
In OO-DB : alle rijen hebben een unieke identificatie (door het systeem)

REF : om identificatie op te vragen
select ref(spelers)
from spelers
where spelersnr = 6 ;

REF : om tabellen te koppelen
create table teams
(teamnr smallint primary key,
speler ref(spelers) not null,
divisie char(6) not null)

insert into teams (teamnr, speler, divisie)
values (3, (select ref(spelers) from spelers where spelersnr = 112), 'ere')

select teamnr, speler.naamfrom teams

Voordelen : ●Altijd het juiste datatype bij de refererende sleutel ●Indien primary keys breed zijn, bespaart het werken met reference-kolommen opslagruimte ●Bij wijzigen van primary keys wordt geen tijd verloren door het wijzigen van de refererende sleutel ●Bepaalde selects worden eenvoudiger **Nadelen :**
●Bepaalde mutaties zijn moeilijker te definiëren ●References werken in één richting ●Bij DB-ontwerp krijgt men meerdere keuzes, dit wordt dus moeilijker ● References kunnen de integriteit van de gegevens niet bewaken zoals refererende sleutels dat kunnen

Collecties → verzameling waardes in één cel
create table spelers
(spelersnr smallint primary key, ...
telefoons setof(char(13)),
bondsnr char(4));
insert into spelers (spelersnr, ..., telefoons, ...)
values (213, .., {'016-342654', '0475-654387'}, ..);
select spelersnr from spelers
where '016-342654' in (telefoons);

select spelersnr from spelers
where cardinality(telefoons) > 2

select TS.telefoons
from the (select telefoons from spelers) as TS
order by 1

Overerving van tabellen → alle eigenschappen van één tabel worden overgeërfd door een andere tabel (supertabel – subtabel)
Beperkingen : ●Geen cyclische structuur ●Geen meervoudige overerving ●Alleen getypeerde tabellen in de tabelhiërarchie ●Overeenkomst tabelhiërarchie met typehiërarchie

create type t_spelers as
(spelersnr smallint not null,
naam char(15) not null, ...
bondsnr char(4))

create type t_oude_spelers as
(vertrokken date not null) under t_spelers

create table spelers of t_spelers
(primary key spelersnr)

create table oude_spelers of t_oude_spelers under spelers

create table spelers as
(spelersnr smallint not null,
naam char(15) not null, ...
bondsnr char(4));

create table oude_spelers
(vertrokken date not null)
inherits (spelers, okra);

SELECT *
FROM (ONLY) spelers;

RULES ●Gaan verder dan triggers : SELECT, INSERT, UPDATE, DELETE ●FKs >> triggered ●Updateble views ●rules of triggers ●Maar voorzichtig, systeemlogica	CREATE RULE "NeKeerIetsAnders" AS ON SELECT TO wedstrijden WHERE spelersnr = 7 DO INSTEAD SELECT 'eerst drie toerkes rond tafel lopen en dan nog eens proberen';

NOSQL

A NoSQL (originally referring to "non SQL") database provides a mechanism for storage and retrieval of data that is modeled in means other than tabular relations used in relational databases. Such databases have existed since the late 60s, but did not obtain the "NoSQL" moniker until a surge of popularity in the 21st century, triggered by the needs of Web 2.0 companies such as Facebook, Google, and Amazon. NoSQL databases are increasingly used in big data and real-time web applications. NoSQL systems are also sometimes called "Not only SQL" to emphasize that they may support SQL-like query languages.

Voorbeelden

●Wide Column Store → Cassandra, Hadoop, HBase, Hypertable, Bigtable ●Document Store →Mongo db, eXist ●Graph based →Neo4J, Sones	●Key Value →Berkeley DB, MemcacheDB ●Multivalue →U2, Reality ●Andere niet sql databases →Versant.., IBM Lotus/Domino

Mongo db ●Binaire JSON ●CRUD ●Index, Aggregation, Replication, Sharding, Needs MapReduce	{"firstName": "John", "lastName": "Smith", "age": 25, "address": {"streetAddress": "21 2nd", "city": "New York", "state": "NY", "postalCode": "10021"}, "phoneNumber": [{"type": "home", "number": "212 555-1234"}, {"type": "fax", "number": "646 555-4567"}]}	**Key Value** Cf Dictionary (ODMS) Postgresql : hstore of gewoon index op key maken	**ACID (transactions)** ●Atomicity: cf boolean ●Consistency: cf state ●Isolation: cf concurrency ●Durability: cf commit, levensduur en select

Some Closing remarks

●Don't use the hammer to fix the plumbing ● Know your tools/options ● Context/Situation ● HandCrafted vs ManHours ● Humans vs Mathematics ● Data independence ●Standarization ● ACID vs Performance
Voor bijna elke klein bedrijf tot KMO is een RDBMS prima.

XML

Extensible Markup Language is een ISO-standaard voor de syntaxis van formele opmaaktalen waarmee men gestructureerde gegevens kan weergeven in de vorm van platte tekst. Deze presentatie is zowel machineleesbaar als leesbaar voor de mens. De afspraken over de te gebruiken tags in de "standaard"-dialecten worden formeel vastgelegd in zogenaamde DTD's (Document Type Definition) of in de nieuwere XML-Schemadefinities (XSD). Je kan attributen aan je tags meegeven, deze moeten tussen “ “ of ' ' staan. Gebruik dit enkel om METAdata weer te geven, bv eigenschappen.

●ISO ●uitwisselbaarheid ●eenvoud ●gn hierarchische engine nodig	**Pro**	**Contra** ●redundantie tov relationeel model, dus niet misbruiken in die zin ●sequentieel, traag

RDBMS-manieren om xml te gebruiken
●SELECT xmlcomment('hallo'); → *xmlcomment/----------/<!--hallo-->*
●SELECT xmlelement(name jos,xmlattributes(current_date as ke), 'hal', 'lo'); →
xmlelement/----------/<jos ke="2014-01-26">hallo</jos>
●SELECT xmlforest(divisie, teamnr) from teams; →
xmlforest/---------/<divisie>ere</divisie><teamnr>1</teamnr>/<divisie>tweede</divisie><teamnr>2</teamnr>
●SELECT xmlpi(name php, 'echo "Patat";'); → *xmlpi/----------/<?php echo "Patat";?>*
●table_to_xml(tbl regclass, nulls boolean, tableforest boolean, targetns text)
●query_to_xml(query text, nulls boolean, tableforest boolean, targetns text)
●cursor_to_xml(cursor refcursor, count int, nulls boolean, tableforest boolean, targetns text)

JSON

JavaScript Object Notation is leesbaarder dan XML zonder parser, qua mogelijkheden slechts subset van XML.

●-> int get element at int position (0 = first elem) ●-> text get element by key text ●->> int get JSON array at int position ●->> text get JSON array by key text ●#> text[] get JSON object at specified path ●#>> text[] get JSON object at specified path as text	→ **Specific Operators** ●'{"a": {"b":{"c": "foo"}}}'::json#>'{a,b}‘ {"c": "foo"} ●'{"a": {"b":"foo"}}'::json→'a {"b":"foo"} ●'[1,2,3]'::json→>2 3

Specific Creation Functions	**Specific Aggregation Functions**	**Specific Processing Functions**
●to_json(element) ●array_to_json(n-dim array) ●json_object (text[]) ●json_object (keys text[], values text[]) ●jsonb_* functions as well	●json_agg(expression) *aggregates values as JSON array* ●json_object_agg(name, value) *aggregates name/value pairs as a JSON object* ●jsonb_* functions as well	●json_array_length(json) ●json_each(json) ~ pop ●json_extract_path(x json, path_elems text[]): get specific elements ●json_array_elements(json): array to values ●jsonb_* functions as well

select * from json_each('{"a":"foo", "b":"bar"}') →	key | value -----+----- a | “foo” b | “bar”	JSONB, decomposed binary format: slower to enter data, faster to process data + allows for indexing, but: ●does not preserve white space, ●does not preserve the order of object keys, ●does not keep duplicate object keys. JSON for the programmer, JSONB for the dba?

RDBMS data types & Other Particular Formats

Buiten bekende ints &c, Reals &c (monetary bv), Strings/Chars &c., alsook JSON en blob. Nog exotischer:

Geometrische types

2d spatiäle objecten. Functies voor schalen, vertalen, roteren en intersecties

Name	Storage Size	Description	Representation
`point`	16 bytes	Point on a plane	(x,y)
`line`	32 bytes	Infinite line	{A,B,C}
`lseg`	32 bytes	Finite line segment	((x1,y1),(x2,y2))
`box`	32 bytes	Rectangular box	((x1,y1),(x2,y2))
`path`	16+16n bytes	Closed path (similar to polygon)	((x1,y1),...)
`path`	16+16n bytes	Open path	[(x1,y1),...]
`polygon`	40+16n bytes	Polygon (similar to closed path)	((x1,y1),...)
`circle`	24 bytes	Circle	<(x,y),r> (center point and radius)

Network Address Types

Name	Storage Size	Description
`cidr`	7 or 19 bytes	IPv4 and IPv6 networks
`inet`	7 or 19 bytes	IPv4 and IPv6 hosts/networks
`macaddr`	6 bytes	MAC addresses

Text Search Types

●**tsvector**: represents a document in a form optimized for text search

●**tsquery**: similarly represents a text query

Meant for natural language processing tasks.
Specific functions & operators

Composite Types	**Range types**	
represents structure of a row or record; it is essentially just a list of field names and their data types. ●CREATE TYPE complex AS (r double precision, i double precision); ●CREATE TYPE inventory_item AS (name text, supplier_id integer, price numeric);	are data types representing a range of values of some element type (called the range's subtype). E.g. ranges of timestamp might be used to represent the ranges of time that a meeting room is reserved. In this case the data type is tsrange (short for "timestamp range"), and timestamp is the subtype. The subtype must have a total order so that it is well-defined whether element values are within, before, or after a range of values.rich set of functions and operators for geometric operations such as scaling, translation, rotation, and determining intersection	●int4range — Range of integer ●int8range — Range of bigint ●numrange — Range of numeric ●tsrange — Range of timestamp without time zone ●tstzrange — Range of timestamp with time zone ●daterange — Range of date