datacharmer.org

SQL bien avancé

inspiré en partie par :

SQL Hacks

Tips & Tools for Digging into Your Data

Andrew Cumming & Gordon Russel

O'Reilly Hacks Series

Fabien Coelho

MINES ParisTech

Composé avec LaTeX, révision numéro 3802

Approche orientée problème

1. valeurs directes
2. énumérer une série
3. requête latérale
4. upsert
5. héritage
6. aggrégation partielle
7. regroupements partiels
8. calculer la valeur médiane
9. générer une somme partielle
10. fermeture transitive
11. SQL est-il Turing complet ?
12. comparer deux tables

Valeurs directes (1)

— construire table constante au vol

— `SELECT ... UNION`

```
SELECT 1 AS jour, mois.nom AS mois
FROM (SELECT 'janvier' AS nom
      UNION SELECT 'février'
      UNION SELECT 'mars'
      UNION SELECT 'avril'
      UNION SELECT 'mai'
      UNION SELECT 'juin'
      UNION SELECT '...') AS mois;
```

jour	*mois*
1	...
1	juin
1	avril
1	février
1	janvier
1	mai
1	mars

Valeurs directes (2)

— `VALUES(...), (...)...`

— aussi utilisé pour `INSERT`

justifie le `S` !

```
SELECT 1 AS jour, mois.nom AS mois
FROM (VALUES
  ('juillet'), ('août'), ('septembre'),
  ('octobre'), ('novembre'), ('décembre'),
  ('...'))
    AS mois(nom);
```

jour	*mois*
1	juillet
1	août
1	septembre
1	octobre
1	novembre
1	décembre
1	...

Valeurs directes (3)

— pas forcément dans une sous requête

```
VALUES (1, 'janvier'), (2, 'février'),
       (3, 'mars'),    (4, 'avril'),
       (5, 'mai'),     (6, 'juin');
```

column1	*column2*
1	janvier
2	février
3	mars
4	avril
5	mai
6	juin

Énumérer une série

— combien de vendredi 13 au 20ème siècle ?

day	*nb*
lundi	173
mercredi	173
samedi	172
dimanche	171
vendredi	171
jeudi	171
mardi	169

Génération d'une relation...

— génération d'intervale avec `generate_series()`

```
SELECT
  CASE EXTRACT(DOW FROM DATE (year || '-' || month || '-13'))
    WHEN 0 THEN 'dimanche' WHEN 1 THEN 'lundi' WHEN 2 THEN 'mardi'
    WHEN 3 THEN 'mercredi' WHEN 4 THEN 'jeudi' WHEN 5 THEN 'vendredi'
    WHEN 6 THEN 'samedi' END AS day,
  COUNT(*) AS nb
FROM generate_series(1901,2000) AS year,
     generate_series(1,12) AS month
GROUP BY day
ORDER BY nb DESC;
```

Tableau constant...

```
SELECT
  -- un tableau constant
  ('{dimanche,lundi,mardi,mercredi,jeudi,vendredi,samedi}'::TEXT[])
    [1 + EXTRACT(DOW FROM DATE (year || '-' || month || '-13'))] AS day,
  COUNT(*) AS nb
FROM generate_series(1901,2000) AS year,
     generate_series(1,12) AS month
GROUP BY day
ORDER BY nb DESC;
```


Requête *à côté* LATERAL

- dans sous-requête FROM
- peut utiliser les valeurs des tuples à sa gauche
- peu utile ? expressions, jointures, aggrégations, fenêtres...

```
-- version avec LATERAL
SELECT i, n AS pow
FROM generate_series(0, 7) AS i,
  LATERAL (SELECT POW(2, i)) AS p2(n);

-- version sans LATERAL
SELECT i, POW(2, i) AS pow
FROM generate_series(0, 7) AS i;
```

i	*pow*
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128

MERGE/UPSERT : INSERT ou UPDATE

- insertion ou mise à jour si existe déjà
  économise un test et sa latence
- différentes syntaxes selon les bases de données...
  MERGE (standard, ...), UPSERT, INSERT ... ON
- PostgreSQL : INSERT ... ON CONFLICT DO NOTHING
  ou INSERT ... ON CONFLICT DO UPDATE ...
  aussi clause WHERE, données initiales EXCLUDED

Exemple *UPSERT*

Données initiales

id	*name*
1	Calvin
2	hbs

```
INSERT INTO Heroes(name) VALUES ('Calvin')
ON CONFLICT (name) DO NOTHING;

INSERT INTO Heroes VALUES (2, 'Hobbes')
ON CONFLICT (id) DO UPDATE SET name=EXCLUDED.name;

INSERT INTO Heroes VALUES (3, 'Susie')
ON CONFLICT (name) DO NOTHING;
```

Données finales

id	*name*
1	Calvin
2	Hobbes
3	Susie

Héritage entre relations

- lien modèles relationnel et objet

```
CREATE TABLE Identité(
  pid SERIAL PRIMARY KEY,
  nom TEXT UNIQUE NOT NULL);

CREATE TABLE Élève(
  promo TEXT NOT NULL,
  UNIQUE(nom)
) INHERITS (Identité);

CREATE TABLE Professeur(
  matière TEXT NOT NULL,
  UNIQUE(nom)
) INHERITS (Identité);
```


Remplissage d'une hiérarchie de tables

```
INSERT INTO Identité(nom) VALUES
  ('Mum'), ('Dad');
INSERT INTO Élève(nom, promo) VALUES
  ('Calvin', '4th'),
  ('Hobbes', '4th');
INSERT INTO Professeur(nom, matière) VALUES
  ('Rosalyn', 'math');
```


Extraction sur une hiérarchie de tables

```
SELECT * FROM Identité;
```

pid	*nom*
1	Mum
2	Dad
3	Calvin
4	Hobbes
5	Rosalyn

```
SELECT * FROM Élève;
```

pid	*nom*	*promo*
3	Calvin	4th
4	Hobbes	4th

Aggrégations partielles FILTER

- applique une aggrégations sur certains tuples seulement
- syntaxe : AGG(...) FILTER(WHERE ...)

```
-- nombre de films avant et après 1950
SELECT
  COUNT(*) FILTER(WHERE année < 1950) AS "avant",
  COUNT(*) FILTER(WHERE année >= 1950) AS "après",
  COUNT(*) AS "total"
FROM films;
```

avant	*après*	*total*
5	6	11

`GROUPING SETS`, `CUBE`, `ROLLUP`

— aggrégation sur des sous-ensembles de `GROUP BY`

`GROUPING SETS` liste de groupes de colonnes

`ROLLUP` tous les préfixes de colonnes, y compris vide

`CUBE` toutes les sous-ensembles de colonnes

— équivalent à `UNION`, valeurs non gardées remplacées par `NULL`

Exemple *grouping sets*

```
CREATE TABLE PaysLanguePopulation(
  id SERIAL PRIMARY KEY,
  pays TEXT NOT NULL,
  langue TEXT NOT NULL,
  pop FLOAT4 NOT NULL,
  UNIQUE (pays, langue)
);
```

pays	*langue*	*pop*
Allemagne	Allemand	81.5
Autriche	Allemand	8.7
Belgique	Allemand	0.1
Belgique	Français	4.1
Belgique	Néerlandais	7.0
France	Français	66.1
Italie	Italien	60.8
Pays-Bas	Néerlandais	16.9
Suisse	Allemand	5.2
Suisse	Français	1.7
Suisse	Italien	0.5

Cumuls par pays et par langues

```
SELECT pays, langue,
       SUM(pop) AS pop
FROM PaysLanguePopulation
GROUP BY GROUPING SETS
  ((pays), (langue), ())
ORDER BY pays, langue;
```

pays	*langue*	*pop*
Allemagne		81.5
Autriche		8.7
Belgique		11.2
France		66.1
Italie		60.8
Pays-Bas		16.9
Suisse		7.4
	Allemand	95.5
	Français	71.9
	Italien	61.3
	Néerlandais	23.9
		252.6

Équivalent avec `UNION`

(probablement moins performant)

```
SELECT pays AS pays, NULL AS langue,
       SUM(pop) AS pop
FROM PaysLanguePopulation
GROUP BY pays -- premier groupe
UNION
SELECT NULL, langue, SUM(pop)
FROM PaysLanguePopulation
GROUP BY langue -- second groupe
UNION
SELECT NULL, NULL, SUM(pop)
FROM PaysLanguePopulation
-- dernier groupe
ORDER BY pays, langue;
```

pays	*langue*	*pop*
Allemagne		81.5
Autriche		8.7
Belgique		11.2
France		66.1
Italie		60.8
Pays-Bas		16.9
Suisse		7.4
	Allemand	95.5
	Français	71.9
	Italien	61.3
	Néerlandais	23.9
		252.6

Exemple `ROLLUP`

```
SELECT pays, langue,
       SUM(pop) AS pop
FROM PaysLanguePopulation
GROUP BY ROLLUP(pays, langue)
-- équivalent à GROUPING SETS
--   ((pays, langue), (pays), ())
ORDER BY langue, pays
LIMIT 12;
```

pays	*langue*	*pop*
Allemagne	Allemand	81.5
Autriche	Allemand	8.7
Belgique	Allemand	0.1
Suisse	Allemand	5.2
Belgique	Français	4.1
France	Français	66.1
Suisse	Français	1.7
Italie	Italien	60.8
Suisse	Italien	0.5
Belgique	Néerlandais	7
Pays-Bas	Néerlandais	16.9
Allemagne		81.5

Exemple `CUBE`

```
SELECT pays, langue,
       SUM(pop) AS pop
FROM PaysLanguePopulation
GROUP BY CUBE(pays, langue)
-- équivalent à GROUPING SETS
--   ((pays, langue),
--    (pays), (langue), ())
ORDER BY pays, langue
LIMIT 13;
```

pays	*langue*	*pop*
Allemagne	Allemand	81.5
Allemagne		81.5
Autriche	Allemand	8.7
Autriche		8.7
Belgique	Allemand	0.1
Belgique	Français	4.1
Belgique	Néerlandais	7
Belgique		11.2
France	Français	66.1
France		66.1
Italie	Italien	60.8
Italie		60.8
Pays-Bas	Néerlandais	16.9

Fonction de fenêtrages *window functions*

— accès aux tuples voisins dans `SELECT`

un peu comme `GROUP BY`, mais sans le regroupement

— numérotation selon un tri, une partition, les deux...

— syntaxe : fonctions et clauses

fonctions d'aggrégation `COUNT SUM`...

fonctions spécifiques `RANK LAG`...

clause FILTER(WHERE ...) filtrage

clause OVER (...) fenêtre de tuples

nommage clause WINDOW ... AS (...) pour réutilisation

Numérotation selon un tri

— syntaxe : `RANK() OVER (ORDER BY ...)`

```
SELECT *,
  RANK() OVER (ORDER BY val DESC)
FROM Notes
ORDER BY id;
```

id	*val*	*rank*
1	10	5
2	15	2
3	17	1
4	13	4
5	15	2

```
SELECT *,
  RANK() OVER (ORDER BY val ASC) AS r1,
  RANK() OVER (ORDER BY val ASC, id ASC) AS r2,
  RANK() OVER (ORDER BY val DESC, id DESC)
    AS r3
FROM Notes
ORDER BY id;
```

id	*val*	*r1*	*r2*	*r3*
1	10	1	1	5
2	15	3	3	3
3	17	5	5	1
4	13	2	2	4
5	15	3	4	2

Regroupement selon une partition

— syntaxe : `AVG(...) OVER (PARTITION BY ...)`

```
SELECT eleve, cours, note,
   -- moyenne par cours
   AVG(note) OVER (PARTITION BY cours)
     AS avc,
   -- moyenne par élève
   AVG(note) OVER (PARTITION BY eleve)
     AS ave
FROM Notations
ORDER BY ave DESC, note DESC;
```

eleve	*cours*	*note*	*avc*	*ave*
Susie	Maths	19	12.25	18.5
Susie	Physics	18	13.25	18.5
Hobbes	Physics	19	13.25	17.5
Hobbes	Maths	16	12.25	17.5
Calvin	Physics	11	13.25	10.5
Calvin	Maths	10	12.25	10.5
Moe	Physics	5	13.25	4.5
Moe	Maths	4	12.25	4.5

Calculer la valeur médiane (le tuple médian)

nom	*delais*
calvin	8
mum	13
dad	7
suzy	10
hobbes	123450

avg
24697.600000000000

nom	*delais*
suzy	10

Aggrégations sur groupes ordonnés

— Fonctions `mode`, `percentile_cont`, `percentile_disc`
`WITHIN GROUP (ORDER BY ...)`

```
SELECT *
FROM AttentePatients
WHERE delais = ( -- attente médiane
  SELECT percentile_disc(0.5) WITHIN GROUP (ORDER BY delais)
  FROM AttentePatients);
```


Générer une somme partielle

id	*quand*	*montant*	*description*	*sum*
1	2005-02-01 00:00:00	1000.00	versement initial	1000.00
2	2005-05-09 00:00:00	-100.00	tirage	900.00
3	2005-05-11 00:00:00	-450.00	sous-tirage	450.00
4	2006-01-01 00:00:00	200.00	ouf	650.00
5	2006-12-23 00:00:00	-700.00	joyeux noel	-50.00
6	2007-02-28 00:00:00	123.45	des sous !	73.45

Jointure avec opérateur $\leq$

— associe *tous* les précédents à chaque opération
— complexité en n^2... plus ou moins inutilisable
— préférer une solution applicative ?

```
-- jointure speciale...
SELECT cc.id, cc.quand, cc.montant, cc.description,
       SUM(run.montant)
FROM CompteCheque AS cc
JOIN CompteCheque AS run ON run.id<=cc.id
GROUP BY cc.id, cc.quand, cc.montant, cc.description
ORDER BY cc.id ASC;
```


Avec une fenêtre – *window*

```
SELECT id, quand, montant, description,
       SUM(montant) OVER (ORDER BY id ASC)
FROM CompteCheque;
```

id	*quand*	*montant*	*description*	*sum*
1	2005-02-01 00:00:00	1000.00	versement initial	1000.00
2	2005-05-09 00:00:00	-100.00	tirage	900.00
3	2005-05-11 00:00:00	-450.00	sous-tirage	450.00
4	2006-01-01 00:00:00	200.00	ouf	650.00
5	2006-12-23 00:00:00	-700.00	joyeux noel	-50.00
6	2007-02-28 00:00:00	123.45	des sous !	73.45

Générer une moyenne mobile

```
SELECT *,
  AVG(qt) OVER (ORDER BY yr
    ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING)
      AS ma
FROM oilprod
ORDER BY yr ASC;
```

yr	*qt*	*ma*
1980	3.1	3.6
1981	4.2	4.2
1982	5.3	5.2
1983	6.2	6.2
1984	7.1	7.1
1985	7.9	7.7
1986	8.1	8.1
1987	8.3	8.1
1988	7.8	7.9
1989	7.6	7.7

Requête *récursive* et fermeture transitive

`WITH` nommage d'une requête temporaire
— sorte de vue locale à une requête, de sous-requête
— calcul indépendant, stockage dans une table temporaire
— utilisation à la suite immédiate
— peut inclure `INSERT`, `UPDATE`, `DELETE`
— attention, barrière d'optimisation : implémentation matérialisée

`WITH RECURSIVE` calcul itératif...
— calcul fermeture transitive : arrêt quand ajout vide

WITH : moyenne inférieure à la moyenne

— réutilisation locale d'une sous-requête...

```
WITH auteur_moy(auteur,moy) AS (
  SELECT nom, AVG(durée)
  FROM Films NATURAL JOIN Personnes
  GROUP BY nom)
SELECT auteur, moy
FROM auteur_moy
WHERE moy < (SELECT AVG(moy) FROM auteur_moy)
ORDER BY auteur;
```

auteur	*moy*
Allen	01:42:00
Ozu	01:37:00

WITH RECURSIVE : fermeture transitive

— table `EnfantDe` des ascendants directs

— calcul des ascendants et leur degré

— stockage table `Ascendant`

```
CREATE TABLE Ascendant(
  enfant TEXT NOT NULL,
  parent TEXT NOT NULL,
  degre INTEGER NOT NULL,
  PRIMARY KEY(enfant,parent)
);
```

enfant	*parent*
Calvin	Dad
Calvin	Mum
Granny	Great Granny
Mum	Grand Pa
Mum	Granny

WITH RECURSIVE : exemple

```
WITH RECURSIVE Ascend(enfant,parent,degre)
AS (
    -- initialisation
    SELECT enfant, parent, 1
    FROM EnfantDe
  UNION
    -- itérations
    SELECT ed.enfant, a.parent, 1+a.degre
    FROM Ascend AS a
    JOIN EnfantDe AS ed ON (ed.parent=a.enfant)
)
-- stocke le résultat dans Ascendant
INSERT INTO Ascendant(enfant, parent, degre)
  SELECT enfant, parent, degre
  FROM Ascend;
```

enfant	*parent*	*degre*
Calvin	Dad	1
Calvin	Mum	1
Granny	Great Granny	1
Mum	Grand Pa	1
Mum	Granny	1
Calvin	Grand Pa	2
Calvin	Granny	2
Mum	Great Granny	2
Calvin	Great Granny	3

WITH RECURSIVE : contraintes

— `WITH RECURSIVE` démarrage

— `SELECT` simple, contenu initial

— `UNION` avec l'incrément d'itération

— `SELECT` sur la table elle-même
qui ne doit apparaître qu'une fois !

— calcul fermeture transitive :
implémentation par une boucle sur une table temporaire

WITH avec INSERT UPDATE DELETE

```
WITH archived AS (
  DELETE FROM Invoice
  WHERE paid AND sent < CURRENT_DATE - INTERVAL '1 month'
  RETURNING *
)
INSERT INTO Archive
SELECT * FROM archived;

WITH changed AS (
  UPDATE stuff
  SET something = 'changed'
  WHERE condition
  RETURNING *
)
SELECT * FROM changed;
```


Plus longues séquences de températures croissantes

http://tapoueh.org/blog/2018/02/find-the-number-of-the-longest-continuously-rising-days-for-a-stock/

— variations journalières
`LAG(c, 1) OVER (ORDER BY d)`

— longueur des séquences croissantes
`WITH RECURSIVE`

— sélection des plus longues `MAX`

date	*temp*
2017-01-01	-0.5
2017-01-02	1.3
2017-01-03	1.1
2017-01-04	1.4
2017-01-05	2.4
2017-01-06	1.1
2017-01-07	-2.2
2017-01-08	-1.3
2017-01-09	-1.5
2017-01-10	2.0


```
WITH RECURSIVE
  Delta AS ( -- day-on-day temperature delta
    SELECT LAG(date, 1) OVER (ORDER BY date) AS dstart,
    date AS dend,
    temp - LAG(temp, 1) OVER (ORDER BY date) AS inc
    FROM Temperature),
  RaisingTemp AS ( -- increasing temperature sequences
    SELECT dstart, dend, 1 AS cnt
    FROM Delta
    WHERE inc >= 0
  UNION
    SELECT p.dstart, d.dend, p.cnt + 1
    FROM RaisingTemp AS p
    JOIN Delta AS d ON (p.dend = d.dstart)
    WHERE d.inc >= 0)
  -- keep longest sequences found
  SELECT dstart, MAX(cnt) AS cnt
  FROM RaisingTemp
  GROUP BY dstart
  ORDER BY cnt DESC, dstart ASC LIMIT 5;
```


PostgreSQL/SQL Turing complet : *oui !*

http://blog.coelho.net/tags.html#Turing-ref

— bande semi infinie de symboles, position, état

— transitions : nouvel état, symbole et position

boucle ...

— `WITH RECURSIVE` pour itérer

— `MAX OVER` pour retrouver le symbole suivant

— `CROSS JOIN` pour agrandir la bande

récursion avec une fonction SQL

Synchroniser deux relations

hypothèses tables avec clef primaire et autres attributs
- — localement (même base de données)
- — à distance (bases hétérogènes)

comparaison trouver et analyser les différences
- — `INSERT`, `UPDATE`, `DELETE`
- — moyen : requête locale, sommes de contrôles hiérarchiques

List of Slides

2 SQL bien avancé
2 SQL Hacks
3 Approche orientée problème
4 Valeurs directes (1)
5 Valeurs directes (2)
6 Valeurs directes (3)
7 Énumérer une série
8 Génération d'une relation...
9 Tableau constant...
10 Requête *à côté* `LATERAL`
11 MERGE/UPSERT : `INSERT` ou `UPDATE`
12 Exemple *UPSERT*
13 Héritage entre relations
14 Remplissage d'une hiérarchie de tables
15 Extraction sur une hiérarchie de tables
16 Aggrégations partielles `FILTER`
17 `GROUPING SETS`, `CUBE`, `ROLLUP`
18 Exemple *grouping sets*
19 Cumuls par pays et par langues
20 Équivalent avec `UNION`
21 Exemple `ROLLUP`
22 Exemple `CUBE`
23 Fonction de fenêtrages *window functions*
24 Numérotation selon un tri
25 Regroupement selon une partition
26 Calculer la valeur médiane (le tuple médian)
27 Aggrégations sur groupes ordonnés
28 Générer une somme partielle
29 Jointure avec opérateur $\leq$
30 Avec une fenêtre – *window*
31 Générer une moyenne mobile
32 Requête *récursive* et fermeture transitive
33 `WITH` : moyenne inférieure à la moyenne
34 `WITH RECURSIVE` : fermeture transitive
35 `WITH RECURSIVE` : exemple
36 `WITH RECURSIVE` : contraintes
37 `WITH` avec `INSERT UPDATE DELETE`
38 Plus longues séquences de températures croissantes
40 PostgreSQL/SQL Turing complet : *oui !*
41 Synchroniser deux relations