Hints para realizar el trabajo final

Qué hacer para el informe final

La idea es comparar el rendimiento de consultas evaluadas por la base de datos de acuerdo al tipo de indización y tamaño de las tablas. Para eso es necesario:

1) Definir consultas complicadas. Normas de dificultas no excluyentes que debieran cumplir las consultas:
    1. Cinco o más tablas en el FROM, donde la consulta es simple.
    2. Cuatro o más tablas, con agregación/agrupación.
    3. Tres o más tablas, con consultas anidadas.
2) Establecer planes de índices, donde cada plan sólo considera un tipo de índices. Las posibilidades:
    1. No usar índices.
    2. Usar árboles B+.
    3. Usar hashing.
3) Considerar tablas de distinto tamaño. Para asegurara una buena detección de costos logarítmicos y lineales, considere:
    1. Tamaños de tablas (número de tuplas/filas) como potencias. Ej: 64, 128, 256, 512, 1024, 2048. Otro ej: 128, 512, 2048, 8132. Otro más: 50, 125, 625, 3125.
    2. Las tablas demasiado pequeñas pueden ser evaluadas en RAM. Cuidado.
    3. Hay tablas de tamaño fijo, ej. una de las regiones de Chile. Esas tablas no se modifican.

Sobre la evaluación del rendimiento, lo que hay que hacer es comparar los tiempos de evaluación de las consultas. Se puede hacer a través de muestreos repetitivos (¡cuidado con que la base de datos use su caché!) o usando consultas como EXPLAIN ANALYZE. Esta última permite saber cómo Postgresql planea la evaluación de una consulta y lo que pasa finalmente. Ejemplo de uso: EXPLAIN ANALYZE SELECT ...

Redacción del informe final

Interesa que indique y justifique las consultas, que además deben ser frecuentes o relevantes en el contexto de la BD (en la medida de lo posible). Una posible organización es:

1) Introducción: lo que viene / el resumen ejecutivo / el contexto / lo que se ha hecho / etc.
2) Las consultas: cada consulta (2 ó 3) / qué hacen / por qué son importantes.
3) Descripción de los experimentos: tamaños de tablas / planes de índices / cómo se obtuvieron los datos (medición de tiempo / EXPLAINs) / tiempo total de la carga / máquina / sistema operativo / etc.
4) Análisis teórico de las consultas: super obviable, puede ir de forma sencilla en las conclusiones.
5) Resultados experimentales: buena presentación; tablas estadísticas / gráficos.
6) Conclusiones: reseñas / discusión / comentarios finales / peroración o palabras de cierre.

7) Anexos: varios...
    1. Obligatorio: una porción del DUMP de Postgresql (¡corto! 2 ó 3 páginas).
    2. Opcional: uno o más scripts de carga.
    3. Recomendado: los EXPLAIN ANALYZE.

Para redactar: sea directo y preciso (breve=bueno, ambiguo=malo). Algunas técnicas de buena redacción son:

1) Sintetizar: ej. “Los elementos redundantes son algo que sobran en la redacción de textos informativos de cualquier naturaleza” => “La redundancia sobra en la redacción de textos informativos”.
2) Precisar: ej. “este plan [de índices] es malo” => “usando este plan [de índices], las consultas tardaron más que usando otros planes”. La precisión aumenta el largo de los textos, pero la preocupación debiera radicar en el gran castigo que la ambigüedad da a la credibilidad.
3) Redundar (!): la redundancia es buena cuando ayuda a reforzar una idea complicada, conectar y provocar emociones en el lector. Estrategias típicas son: el uso de ejemplos y casos (si no se usan como *evidencia*), y el uso de “importante:”, “nótese que:” u “ojo:” (no aportan información). Otra táctica es el uso de preguntas: “¿Qué hacemos en este caso? Bueno, una solución...”. El ejemplo anterior sintetizado es: “Una solución para este caso es...”, pero es una redacción áspera. La decisión de cómo redundar depende del trato al lector. Afortunadamente, la síntesis pura no es nociva ya que reduce fuertemente el largo de un texto, y eso se suele agradecer. (Excepción: las discusiones suelen redundar para contrastar puntos.)
4) Composición de párrafos: la primera oración declara la intención y lo que sigue (ej. “Ningún plan de consulta supera a otro en todos los casos.”), luego siguen los detalles de mayor a menor importancia (si es que se pueden ordenar por importancia) que soportan la primera oración.
    1. Excepción: en el último párrafo, las oraciones se hacen cada más importantes.
5) Organización de la información: son estrategias que son válidas según la situación, y deben usarse con consistencia. En general son órdenes según:
    1. Precisión: de lo general al detalle y viceversa.
    2. Importancia: de mayor a menor relevancia y viceversa.
    3. Dependencia lógica: conclusión y sus argumentos, o argumentos y su conclusión. Los argumentos tienen un orden único: bases y producción o inferencias sucesivas.
    4. Cronología: de antes a después, o el final más el recuento. El recuento es siempre de antes a después.
    5. Atractivo literario: de mayor a menor atractivo. (Chocante, polémico, intrigante, curioso, increíble, sorprendente, hacia el climax.)
    6. Órdenes mixtos: ej. entre-párrafos: importancia, intra-párrafos: precisión.
    7. Temas-subtemas: se pueden hacer switchs. Por ejemplo:
        1. Orden 1:
            1. Qué hace cada una
                1. Consulta 1
                2. Consulta 2

    2. SQL
        1. Consulta 1
        2. Consulta 2
    3. Importancia
        1. Consulta 1
        2. Consulta 2
2. Orden 2 (reduce la dependencia entre párrafos => más fácil de leer):
    1. Consulta 1
        1. Qué hace
        2. SQL
        3. Importancia
    1. Consulta 2
        1. Qué hace
        2. SQL
        3. Importancia

Utilizando Postgresql

Veamos cómo configurar una cuenta local de Postgresql (linux/unix):

1) Definir un directorio y asignar una variable de ambiente (PGDATA) para el directorio. Por ejemplo, en bash: `$ export PGDATA=/home/cuentausuario/bd/`. Ojo que el directorio debe existir.
2) Inicializar el directorio para su uso por la BD usando initdb: `$ initdb`
3) Ahora, la base de datos estará vacía, sin bases de datos ni nada. Luego, no se puede utilizar. Pero sí se puede hacer andar. Entonces, ejecutemos la base de datos en el fondo: `$ postmaster &`
4) Postgresql está ejecutándose en el fondo. Es hora de crear una base de datos: `$ createdb`
5) Por fin tenemos una base de datos. Ahora podemos acceder como clientes: `$ psql`
6) Si hemos dejado de usar la base de datos, matemos la BD: `$ kill postmaster`

La segunda vez que ejecutemos Postgresql ya tendremos la BD creada. Luego, lo que habrá que hacer es:

1) Colocar la variable ambiente: `$ export PGDATA=/home/cuentausuario/bd/`
2) Iniciar el demonio Postgresql: `$ postmaster &`
3) Abrir el cliente: `$ psql`
4) Matar el demonio (si no se va a volver a usar la BD): `$ kill postmaster`

Para hacer el DUMP de Postgresql, con el demonio en ejecución: `$ pg_dump`

Instrucciones útiles como cliente

Por ejemplo, ya mencionamos EXPLAN ANALYZE. Esta consulta puede ser esencial para la realización de los experimentos.

La siguiente instrucción esencial es la creación de índices. Por ejemplo:

```
CREATE INDEX tabla1_indice1 ON tabla1(atrib1) USING HASH
```

Los índices a usar son HASH y BTREE. También se pueden indizar varios atributos con BTREE, pero tratemos de evitar eso.

Para eliminar un índice, usaremos la instrucción:

```
DROP INDEX tabla1_indice1
```

Listos con las consultas básicas, pasemos a las instrucciones de *scripts*. En Postgresql podemos ejecutar scripts con la simple instrucción \i:

```
\i script.sql
```

Un script es un archivo de texto plano que contienen instrucciones SQL, y en Postgresql tienen acceso a las instrucciones propias como \i nuevamente. Por ejemplo:

```
gran_script.sql
---------------------
\i crear_tablas.sql
\i insertar_100.sql
\i explains.sql
\i borrar_tablas.sql
```

Llenando tablas

Los archivos de script de creación de tablas son sencillos. Por ejemplo, veamos cómo hacer una tabla y ponerle llaves:

```
CREATE TABLE tableta (id INTEGER, dinero BIGNUM, texto VARCHAR);

ALTER TABLE tableta ADD PRIMARY KEY (id);
```

Y para insertar, usamos:

```
INSERT INTO tableta VALUES(101, 12192989832392, 'asi nomah!!!!!');
```

Si no insertamos los atributos en orden (de definición en el CREATE TABLE), o estamos inseguros:

```
INSERT INTO tableta(id,dinero,texto) VALUES(1, 1219, 'remacanudisimo!!!!');
```

Yap, ahora el problema del estudiante es cómo llenar tablas con cientos o miles de tablas y se cumpla que los join naturales tengan selectividad entre 1% y 10%, o los WHERE de atributos simples cumplan con eso.

El problema de la selectividad de los join natural es sencilla. Por ejemplo, supongamos que las llaves entre tres tablas A, B y C cumplen con:

$$|A| = 10000\,,\, |B| = 30000\,,\, |C| = 90000$$
$$|\{A.id\} \cap \{B.id\}| = 1000$$
$$|\{A.id\} \cap \{B.id\} \cap \{C.id\}| = 100$$

¿Qué *id* debemos asignar a esas tablas? Da lo mismo la generación, y suponiendo que los *id* son únicos, podemos darles números ordenados. Por ejemplo:

$$\{A.id\} = [1,100] \cup [1001,1900] \cup [10001,20000-1000]$$
$$\{B.id\} = [1,100] \cup [1001,1900] \cup [30001,60000-1000]$$
$$\{C.id\} = [1,100] \cup [70001,160000-100]$$

La elección anterior, de ejemplo, es una de las tantas soluciones al problema de las intersecciones. Y ojo, el problema de las intersecciones está subdeterminado: una buena solución requiere 7 restricciones, y sólo se dieron 5. Ahora, no hay para qué complicarse demasiado; ahí queda hacer soluciones simples.

¿Cómo llenamos la tabla A según la situación anterior? Si A=A(id,atr1,atr2), podemos hacer, en Perl:

```
test.pl
-------------------------------------------------------------------------------
#!/usr/bin/perl
@atr=('a','b','c','d','e');
for($i=1;$i<=100;$i++) {$j=@atr[$i%5]; print “insert into A values($i,'$j','$j');\n”;}
for($i=1001;$i<=1900;$i++) {$j=@atr[$i%5]; print “insert into A values($i,'$j','$j');\n”;}
for($i=10001;$i<=19000;$i++) {$j=@atr[$i%5]; print “insert into A values($i,'$j','$j');\n”;}
```

Ahora, para guardar esto en una tabla, hacemos:

```
$ perl test.pl > insertarA-10000.sql
```

El script de Perl es fácil de traducir a Java. De hecho, ¡queda casi igual! Pero en Perl no hay que compilar; ahí queda elegir lo más cómodo.

Si uno quiere hacer un A de 30000 tuplas, el script Perl queda como:

```
test.pl
-------------------------------------------------------------------------------
#!/usr/bin/perl
@atr=('a','b','c','d','e');
for($i=3;$i<=300;$i++) {$j=@atr[$i%5]; print “insert into A values($i,'$j','$j');\n”;}
for($i=3001;$i<=5700;$i++) {$j=@atr[$i%5]; print “insert into A values($i,'$j','$j');\n”;}
for($i=30003;$i<=57000;$i++) {$j=@atr[$i%5]; print “insert into A values($i,'$j','$j');\n”;}
```

Esta es la conveniencia de usar atributos numéricos y definirlos por intervalos. Para colocar atributos aleatorios, use la instrucción numérica **rand(N)**. Generará un número entre 0 y N-1. Si quiere restringir a enteros, **int(rand(N))**.

Recomendación final: no hay que complicarse demasiado. Lo que interesa es comparar distintos tipos de índices para una situación fija. No interesa definir esa situación demasiado bien; **puede ser cualquiera**. ¡Pero **no haga conjuntos disjuntos**!

Llenado perfecto (ANEXO: NO RECOMENDADO)

Si su curiosidad intelectual es fuerte, invente una solución por su cuenta. Pero si no quiere darse el trabajo y desea hacer algo complicado pero perfecto, lea lo siguiente.

Sea ABC la notación para $|A \cap B \cap C|$ y A' la notación para el complemento de A. Con esta notación, la cardinalidad A es:

$$A = ABC + ABC' + AB'C + AB'C'$$

Ojo que cada trío es disjunto del resto (luego + sirve como *unión*). De hecho, el caso anterior denota una partición de A. ¿Cuántos tríos necesitamos? Si consideramos la construcción de A, B y C, el único trío que no necesitamos es A'B'C'. Luego, necesitamos 7 tríos ($2^3 - 1$). Entonces, si tenemos un sistema lineal de intersecciones, necesitamos un sistema de 7 ecuaciones linealmente independientes.

Por ejemplo, podemos definir un sistema perfecto con lo siguiente:

A=1000, B=1000, C=1000
AB=500, BC=200, AC=100
ABC=50

Y si sabemos traducir lo anterior, escribimos el sistema:

A=1000: ABC+ABC'+AB'C+ABC' = 1000
B=1000: ABC+ABC'+A'BC+A'BC' = 1000
C=1000: ABC+AB'C+A'BC+A'B'C = 1000
AB=500: ABC+ABC' = 500
BC=200: ABC+A'BC = 200
AC=100: ABC+AB'C = 100
ABC=50: ABC = 50

Ojo en hacer este sistema consistente.

Al final, lo único que faltaría es definir intervalos disjuntos para cada trío, con el tamaño que resuelve el sistema lineal. Luego, se arma cada tabla A, B y C de acuerdo a las llaves establecidas.

El tamaño variable de las tablas se hace como se indicó en los scripts Perl anteriores.

¿Cómo podemos definir los intervalos? Con valores contiguos, por ejemplo. O con valores que nos aseguren una definición simple. Por ejemplo, si |A|,|B|,|C| = 1000. Los intervalos pueden ser:

{ABC} = {1000 ... 1000+ABC}
{ABC'} = {2000 ... 2000+ABC'}
{AB'C} = {3000 ... 3000+AB'C}
{AB'C'} = {4000 ... 4000+AB'C'}
{A'BC} = {5000 ... 5000+A'BC}
...

¿Cómo generar tablas de distinto tamaño? Hay que ponderar los extremos de los intervalos.

¿Cómo resolver un sistema incompletamente definido? Para lograr una solución particular de forma automática, se puede usar programación lineal, maximizando el tamaño de algún trío.

¿Si usamos más tablas? Hay que aumentar el número variables. Para *N* tablas, hay que usar $2^N - 1$ variables...