FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Sistemas de Bases de Dados

Mestrado Integrado em Engenharia Informática - Ano Lectivo 2013/14

Grupo 28:

Filipe Correia Nº41873
João Vitorino Nº34274
Tiago Santos Nº41852

Indice

Índice

Introdução

Este trabalho é desenvolvido no âmbito da cadeira de sistemas de gestão de base de dado inserida no Mestrado Intregrado de Engenheiria Informática e visa, como tal, elaborar uma análise do sistema sobre cada uma das vertentes estudadas na disciplina nomeadamente o Armazenamento e file structure, Indexação e hashing, Processamento e optimização de perguntas, Gestão de transacções e controlo de concorrência e Suporte para bases de dados distribuídas com vista na implementação dos conceitos estudados, os algoritmos que esta implementa dizendo quais os que o utilizador pode escolher, quais os comandos para parameterização, etc…
Para além dos capitulos relacionados com a matéria estudada será ainda feito um estudo acerca do tipo de ferramentas que vêm com este, o tipo de serviços que fornece e features adicionais. Por fim será feita uma comparação com o Oracle 11g.

Quanto ao sistema estudado, na primavera de 2000, D. Richard Hipp estava a desevolver um software para destruidores de misseis em contrato com US Navy, sistema este originalmente baseado em HP-UX com suporte a uma base de dados IBM Informix. O objetivo primário do SQLite foi permitir que o programa fosse dirigido sem ser necessária a instalação de um sistema de gestão de base de dados ou ser requerido um administrador. Em Agosto de 2000 foi lançada a versão 1.0 do SQLite com base em gbdm (GNU Database Manager). O SQLite é uma biblioteca em C, in-process que implementa uma base de dados transicional SQL auto-suficiente, e livre de servidor e configuração. O código para SQLite é open-source, portanto, o seu uso serve tanto para fins privados como comerciais.Este é encontrado atualmente em mais aplicações do que podemos contar, incluindo vários projetos de alto perfil.

Armazenamento e file structure

Buffer Management

O SQLite tem um subsistema de alocação dinâmica de memória que utiliza para guardar objectos, construir uma cache para o ficheiro da base de dados e guardar os resultados de queries. Por defeito, utiliza as funções malloc(), realloc() e free() da biblioteca da linguagem C, e a não ser que seja necessário para casos especiais, é recomendado utilizar este memory allocator. No entanto, existem alternativas disponíveis. Para as utilizar, o SQLite deve ser compilado com a opção desejada, algumas delas enumeradas abaixo.

SQLITE_MEMDEBUG: Este memory allocator é destinado a efectuar testes e debug do SQLite, e afecta a performance da aplicação. Não é indicado para efeitos de produção. Os testes realizados pelo memdebug verificam que o SQLite não efectua escritas fora dos limites dos espaços de memória alocados, que blocos de memória livres, depois da chamada a free(), não são utilizados, e que todas as chamadas a free() foram efectuadas sobre blocos de memória previamente alocados com malloc() prévio.

SQLITE_ENABLE_MEMSYS5: O memsys5, também referido como zero-malloc memory allocator é uma alternativa que não utiliza a função malloc() do C. Mesmo quando é incluído na build compilada, o memsys5 está desabilitado por defeito, portanto é preciso fazer a chamada a sqlite3_config(SQLITE_CONFIG_HEAP, pBuf, szBuf, mnReq); em que pBuf é um espaço de memória contíguo que o memsys5 utiliza para todas as alocações de memória.

SQLITE_ZERO_MALLOC: Serve apenas como um stub para substituir o allocator por defeito, em sistemas que não têm as funções malloc(), ou realloc(), ou free() nas suas bibliotecas standard. Aplicações que sejam compiladas com esta opção devem especificar um allocator alternativo invocando a função sqlite3_config().

Abaixo são descritas algumas funcionalidades do subsistema de alocação de memória do SQLite:

Se uma alocação de memória falhar, o SQLite tenta libertar memória das páginas de cache, e de seguida tenta fazer o request de alocação de memória novamente. Se esta tentativa também falhar, o SQLite retorna o error code SQLITE_NOMEM, ou tentará prosseguir sem a memória alocada.

A aplicação é responsável por destruir todos os objectos que aloca, para garantir que não ocorrem memory leaks.

Através do mecanismo sqlite3_soft_heap_limit64(), impõe um limite de utilização de memória que o SQLite tenta não ocupar inteiramente, reutilizando memória da sua cache ao invés de alocar mais memória quando a utilização se aproxima do limite.

De forma a reduzir os tempos de processamento, as chamadas a malloc() e free() são minimizadas.

Virtual File System Objects (VFS Objects)

Os objectos VFS são a interface de comunicação entre o SQLite e o sistema operativo. Quando algum dos módulos do SQLite necessita de comunicar com o sistema operativo, invoca métodos no VFS, que por sua vez invoca o código do sistema operativo correspondente à operação pedida. Isto permite que o SQLite seja portável entre diferentes sistemas operativos, para os quais se têm diferentes VFS. A source tree standard do SQLite contém VFS para sistemas UNIX e Windows, e portá-lo para outros sistemas é apenas uma questão de escrever um novo VFS.

As builds para sistemas UNIX vêm com múltiplos VFS, cujas únicas diferenças são o modo como gerem o file locking. São identificados por nomes unívocos, sendo o VFS utilizado por defeito chamado "unix". Alternativas possíveis ao default são:

"unix-dotfile" - utiliza dotfile locking, em vez de locks POSIX;

"unix-excl" - obtém locks exclusivos sobre ficheiros da base de dados, impedindo que outros processos acedam à base de dados;

"unix-none" - não obtém quaisquer locks, o que resulta facilmente na corrupção da BD se existirem acessos concorrentes.

Embora seja possível utilizar múltiplos VFS simultaneamente, é recomendado usar apenas "unix" e "unix-excl", porque à excepção destes dois, os diferentes VFS utilizam diferentes tipos de locks. Como consequência, se dois processos acederem à base de dados simultaneamente com diferentes VFS, podem não ver os outros locks e interferir, acabando por corromper a base de dados.

O VFS default pode ser alterado, registando o VFS desejado com a chamada a sqlite3_vfs_register(). Por exemplo, para utilizar sempre o VFS "unix-none"

Implementação de um VFS

Para implementar um novo VFS é necessário construir subclasses dos objectos sqlite3_vfs, sqlite3_io_methods e sqlite3_file.

Um objecto sqlite3_vfs define o nome do VFS, os métodos que implementam a interface para o sistema operativo, como verificar a existência de ficheiros, criar,

apagar, ler e escrever ficheiros, e para outras operações do sistema, como suspender processos, ou obter a data actual do sistema.

Um objecto sqlite3_file representa um ficheiro aberto. Este objecto é construído quando o método xOpen, do sqlite3_vfs, é invocado, e mantém o estado do ficheiro aberto.

Um objecto sqlite3_io_methods contém os métodos que interagem com um ficheiro aberto, como ler e escrever sobre o ficheiro, descobrir o seu tamanho, obter locks e fazer unlock do ficheiro, e fechar ou destruir o objecto sqlite3_file. Cada objecto sqlite3_file tem um apontador para um objecto sqlite3_io_methods apropriado para o ficheiro que representa.

Por fim, o objecto sqlite3_vfs tem de ser registado através da chamada a sqlite3_vfs_register().

Organização de tuplos numa base de dados

Uma base de dados SQLite é mantida num único ficheiro, composto por páginas de tamanho uniforme. Os tuplos de uma relação podem estar organizados de formas diferentes (sequência de introdução, hash, organizados por uma chave) e cada relação pode ter uma organização de tuplos diferente. O SQLite utiliza uma B+-Tree para organizar os tuplos de uma relação, e cada relação é organizada por uma B+-Tree diferente, enquanto um índice de uma relação é organizado por uma B-Tree, e a cada índice corresponde uma B-Tree diferente. Estas àrvores estão guardadas num único ficheiro, espalhadas por diferentes páginas, e são a única forma de organização de tuplos de uma base de dados SQLite. Portanto, podemos afirmar que uma base de dados é uma colecção de B+Trees e B-Trees.

Indexação e hashing

O SQLite apenas permite indexação por árvores B+ e índices sob múltiplos atributos, no entanto não implemanta indexação bitmap, que seria preferivel quando se trata de colunas de baixa cardinalidade no que toca a termos espaço e velocidade, nem implementa índices em hash que são preferiveis quando tratamos de colunas de alta seletividade em termos de espaço e acessos de leitura e escrita em disco embora tenha o efeito colateral a recuperação in_order da coluna não poder ser utilizada como indice.

Quanto às árvores B, o SQLite implementa duas destas. O Algoritmo que Knuth se refere como ***B*-Trees*** que armazena os dados nas folhas é intitulado aqui como ***Table B-Tree***. O algoritmo apenas intitulado de ***B-Tree*** e que armazenam tanto a chave juntamente com os dados em folhas e em páginas interiores, é intitulado de ***Index B-Tree*** e na implementação do SQLite apenas a chave é armazenada sendo os dados completamente omitidos. Em relação aos índices sob vários atributos o SQLite permite que sejam criados vários sob o mesmo conjunto de colunas.

Relativamente a organização de ficheiros o SQLite, no estado final da sua base de dados, contem normalmente apenas um ficheiro em disco intitulado de ***"main database file"***. Durante a transação o SQLite armazena a informação adicional num ficheiro secundário chamado "***rollback journal***" ou num log file (caso esteja em modo ***Write-Along Logging***). Se a aplicação ou o host rebentar antes da transação estar concluída, o *rollback journal* ou o *write-ahead log* irá conter as informações de estado necessárias para restaurar o ficheiro da base de dados para um estado consistente. Quando *rollback journal* ou o *write-ahead log* contém as informações necessárias para a recuperação estes são chamados de "***hot journal***" ou "***hot WAL file***". Estes são apenas um fator em cenários de recuperação de erros e por isso são incomuns, mas visto fazerem parte do estado da base de dados SQLite não podem ser ignorados.

Indexações do tipo Clustered não são suportadas no SQLite, isto significa que se o nosso índice for uma sequência de Inteiros, os registos criados na Base de Dados são colocados por essa ordem, primeiro o 1, depois o 2, o 3… Embora não possamos criar índices deste género, podemos ordenar os nossos dados de maneira a que fiquem corretamente ordenados por ordem histórica. Imaginemos que temos uma tabela 'A' onde queremos aceder várias vezes ao campo KEY, seria bom que tudo estivesse ordenado. Através da linha de comando é fácil criar uma espécie de *cluster* falso ao fazer o seguinte:

```
create table A2 as select * from A;
delete from A;
insert into A select * from A2 order by key;
drop table A2;
```

De notar que apesar de ajudarem, as indexações não devem ser utilizadas em casos de tabelas pequenas, tabelas com elevadas quantidades de operações insert e update, colunas com um número elevados de valores NULL e cem colunas que são frequentemente manipuladas.

Seguem-se os **comandos** que envolvem indices em SQLite:

Sintax básica:

```
CREATE INDEX index_name ON table_name;
```

Single-Column Indexes - é aquele que é criado apenas com base numa coluna da tabela. A sintax basica é a seguinte:

```
CREATE INDEX index_name
ON table_name (column_name);
```

Unique Indexes - não são usados apenas para performance, mas também para integridade de dados. Um unique index faz com que não seja permitido que nenhum valor duplicado seja inserido na tabela. A sintaxe básica é a seguinte:

```
CREATE UNIQUE INDEX index_name
on table_name (column_name);
```

Composite Indexes - é um indice com base um duas ou mais colunas de uma tabela. A sintaxe básica é a seguinte:

```
CREATE INDEX index_name
on table_name (column1, column2);
```

De notar que quer em indices simples ou compostos podemos utilizar as colunas com acesso mais frequente como filtros na cláusula WHERE.

Implicit Indexes – são indices que são criados automaticamente pelo DB Server quando um objeto é criado. Os indices são criados automaticamente para primary key constraints e unique constraints.

Consideremos a COMPANY TABLE:

```
DROP TABLE COMPANY;
CREATE TABLE COMPANY(
   ID INT PRIMARY KEY     NOT NULL,
   NAME           TEXT    NOT NULL,
   AGE            INT     NOT NULL,
   ADDRESS        CHAR(50),
   SALARY         REAL
);
```

```
INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (1, 'Paul', 32, 'California', 20000.00 );

INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (2, 'Allen', 25, 'Texas', 15000.00 );

INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (3, 'Teddy', 23, 'Norway', 20000.00 );

INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (4, 'Mark', 25, 'Rich-Mond ', 65000.00 );

INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (5, 'David', 27, 'Texas', 85000.00 );

INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (6, 'Kim', 22, 'South-Hall', 45000.00 );

INSERT INTO COMPANY VALUES (7, 'James', 24, 'Houston', 10000.00 );
```

Sintax para criação de um indice na coluna dos salários:

```
sqlite> CREATE INDEX salary_index ON COMPANY (salary);
```

Para listar todos os indices da tabela COMPANY utiliza-se o comando **.indices**:

```
sqlite> .indices COMPANY
```

Isto irá produzir o seguinte resultado onde *sqlite_autoindex_COMPANY_1* é um indice implicito, o qual foi criado quando a própria tabela se criou.

```
salary_index
sqlite_autoindex_COMPANY_1
```

Podemos ainda listar todos os indices da base de dados da seguinte forma:

```
sqlite> SELECT * FROM sqlite_master WHERE type = 'index';
```

Comando DROP INDEX:

An index can be dropped using SQLite **DROP** command. Care should be taken when dropping an index because performance may be slowed or improved.

Um indice pode ser facilmente removido em SQLite com o comando **DROP**, claro está que há que ter em conta se a operação nos conduz a uma performance reduzida ou melhorada.

A sintax basica é a seguinte:

```
DROP INDEX index_name;
```

Por exemplo para remover o indice anteriormente criado:

```
sqlite> DROP INDEX salary_index;
```

INDEX BY:

A cláusula INDEXED BY pode ser utilizada com as operações DELETE, UPDATE ou SELECT:

```
SELECT|DELETE|UPDATE column1, column2...
INDEXED BY (index_name)
table_name
WHERE (CONDITION);
```

Consideremos a tabeka COMPANY, irá ser criado um indice para a qual se utilizará a operação INDEXED BY:

```
sqlite> CREATE INDEX salary_index ON COMPANY(salary);
sqlite>
```

Agora selecionamos os dados da tabela COMPANY e utilizamos a clausula INDEXED BY da seguinte forma:

```
sqlite> SELECT * FROM COMPANY INDEXED BY salary_index WHERE salary >
5000;
```

Após esta operação obtemos o seguinte resultado:

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
7           James       24          Houston     10000.0
2           Allen       25          Texas       15000.0
1           Paul        32          California  20000.0
3           Teddy       23          Norway      20000.0
6           Kim         22          South-Hall  45000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
```

Processamento e optimização de perguntas

Para o processamento de perguntas existem duas estruturas de dados fundamentais na API, a connection e o statement. Na API C correspondem diretamente ao sqlite3 e ao sqlite3_smt respetivamente. Todas as principais operações da API são realizadas usando uma destas duas estruturas. A connection representa uma conexão à base de dados e um contexto de transação. Os statements são derivados das conexões e representam um statement SQL já compilado. Incluem ainda recursos que mantêm o estado da VDBE à medida que é executado cada passo. Apesar de conterem muitas coisas podemos simplesmente tratá-los comos cursors com os quais iterar o resultado ou handles que referenciam um único comando SQL. Cada conexão pode ter vários objetos da base de dados e cada um desses objetos tem apenas um object da B-tree que por sua vez apenas tem pager. Os statements usam a B-tree e o Pager da sua conexão para ler e escrever dados na base de dados. Para ler iteram a B-tree usando cursors que iteram sobre registos que estão guardados em páginas. À medida que o cursor atravessa registos atravessa também páginas. Para tal a página tem primeiro de ser carregada para memória. Para tal, o pager, sempre que a B-tree necessita de uma página em específico da base de dados carrega a mesma para um buffer em memória. Se o cursor modificar a página o pager tem de tomar medidas para preservar a página original em caso de rollback.

Existem dois métodos essenciais de execução de comandos SQL no SQLite: prepared queries e wrapped queries. Prepared queries são a principal forma que o SQLite usa para executar todos os comandos, quer internos quer da API. O processamento de prepared queries é um processo de três fases: preparação, execução e finalização. Para tal existe uma instrução correspondente com cada fase na API. Associadas ainda à fase de execução estão funções de obtenção de registos e de informação sobre colunas dos resultados. Para além do método standard de query há duas funções de agregação que juntam as três fases numa só função fornecendo uma forma conveniente de executar um comando SQL de uma só vez.

Como dito anteriormente, executar um comando SQL é um processo de três fases:

Preparação: O parser, o tokenizer e o gerador de código compilam o statement para o byte code da VDBE. Na API C isso é feito usando a função sqlite3_prepare_v2(), que "fala" diretamente com o compilador que cria um sqlite3_stmt handle que contem o byte code e todos os outros recursos necessários para a execução do comando e iteração sobre o resultado (se produzido pelo comando).

Execução: A VDBE executa o byte code. A execução é um processo realizado por passos. Na API C cada passo é iniciado com sqlite3_step(), que faz a VDBE avançar no código. Normalmente a primeira chamada do sqlite3_step() adquire um lock que varia consoante o comando (leitura ou escrita). Para statements SELECT cada chamada do sqlite3_step() posiciona o cursor na próxima linha dos resultados. Para cada linha no resultado devolve SQLITE_ROW até chegar ao fim onde devolve SQLITE_DONE. Para outro tipo de statements (insert, update, delete...) a primeira chamada de sqlite3_step() faz a VDBE processar o comando todo.

Finalização: A VDBE fecha o statement e liberta os recursos. Na API C isto é realizado pela função sqlite3_finalize() que faz com que a VDBE termine o programa, liberte os recursos e feche o statement handle.

Cada fase corresponde ao respetivo estado do statement handle (prepared, active ou finalized). Prepared significa que todos os recursos foram alocados e que o statement está pronto a ser executado mas ainda nada começou, nenhum lock foi adquirido nem será até a primeira chamada da função sqlite3_step(). O estado active começa com a primeira chamada da função sqlite3_step(). Por esta altura o statement está no processo de ser executado e já existe algum lock. Finalized significa que o statement foi fechado e todos os recursos associados foram libertados.

O pseudo-código que se segue ilustra o processo geral de execução de uma query no SQLite:

```
#1. Abrir a base de dados e criar um objecto de conexão (db)
db = open('foods.db')
#2.A. Preparar um statement
stmt = db.prepare('select * from episodes')
#2.B. Executar. Chamar step() até o cursor chegar ao fim dos
resultados.
while stmt.step() == SQLITE_ROW
print stmt.column('name')
end
# 2.C. Finalizar. Libertar o lock de leitura.
stmt.finalize()
# 3. Inserir um registo
stmt = db.prepare('INSERT INTO foods VALUES (…)')
```

```
stmt.step()
stmt.finalize()
#4. Fechar a conexão à base de dados.
db.close()
```

Statements SQL podem conter parâmetros. Parâmetros são placeholders nos quais os valores podem ser dados mais tarde durante a execução:

```
insert into foods (id, name) values (?,?);
insert into episodes (id, name) (:id, :name);
```

Estes statements representam duas formas de "binding" de parâmetros: por posição e por nome. O primeiro statement usa parâmetros por posição enquanto que o segundo usa por nome. Parâmetros por posição são definidos pela posição do ponto de interrogação no statement: o primeiro tem a posição 1, o segundo a posição 2, e por ai adiante. Parâmetros por nome usam os nomes de variáveis que são prefixados por dois pontos. Quando o comando sqlite3_prepare_v2() compila um statement com parâmetros aloca placeholders para os parâmetros no statement handle resultante. Fica depois a espera que os valores para estes parâmetros sejam fornecidos antes da execução do statement. Se não for fornecido um valor para um parâmetro o SQLite usa NULL como valor por defeito quando executa o statement. A vantagem do uso de parâmetros é que se pode executar o mesmo statement várias vezes sem ser necessária a recompilação do mesmo. Apenas se tem de fazer reset ao statement, fornecer um novo conjunto de valores e reexecutar evitando assim a compilação. O reset de um statement está implementado na API com a função sqlite3_reset().

O pseudo-código que se segue ilustra o processo básico do uso de parâmetros:

```
db = open('foods.db')
stmt = db.prepare('insert into episodes (id, name) values (:id,
:name)')
stmt.bind('id', '1')
stmt.bind('name', 'Soup Nazi')
stmt.step()
# Reset e usa novamente
stmt.reset()
stmt.bind('id', '2')
stmt.bind('name', 'The Junior Mint')
# Done
stmt.finalize()
db.close()
```

Isto pode melhor significativamente a performance de queries repetitivas já que o compilador deixa de ser necessário a repetição.

Como mencionado antes, há duas funções muito uteis que envolvem o processo de prepared queries, permitindo executar comandos SQL numa só chamada. Uma é normalmente usada para queries que não devolvem nada (sqlite3_exec()). A outra para queries que devolvem (sqlite3_get_table()). A função exec() á uma forma rápida e fácil de executar insert, update, e delete ou statements DDL para criar ou destruir objectos base de dados. Funciona diretamente sobre a conexão à base de dados, usando um handle sqlite3 para uma base de dados aberta em conjunto com uma string contendo um ou mais statements SQL. Internamente o exec() faz parse da string SQL, identifica os statements e processa-os um por um. Aloca os seus próprios statement handles e prepara, executa e finaliza cada statement. Se forem passados vários statements e um deles falhar a função termina a execução nesse comando e devolve o código de erro associado.

O pseudo-código que se segue ilustra o funcionamento da função exec:

```
db = open('foods.db')
db.exec("insert into episodes (id, name) values (1, 'Soup Nazi')")
db.exec("insert into episodes (id, name) values (2, 'The Fusilli
Jerry')")
db.exec("begin; delete from episodes; rollback")
db.close()
```

A segunda função, sqlite3_get_table(), funciona de maneira muito semelhante com a função exec() mas retorna o set completo de resultados em memória.

O pseudo-código que se segue ilustra a forma mais típica de utilização da mesma:

```
db = open('foods.db')
table = db.get_table("select * from episodes limit 10")
for i=0; i < table.rows; i++
      for j=0; j < table.cols; j++
              print table[i][j]
      end
end
db.close()
```

O ponto forte da get_table() é que fornece um método de um passo para realizar uma query e obter resultados. O ponto fraco é que guarda esses resultados completamente em memória, impossibilitando o seu uso com um set de resultados maior. Já o método de prepared query apenas mantem um registo em memória de cada vez, sendo muito melhor para grandes conjuntos de resultados.

Optimização:

Dado um statement SQL podem existir várias formas de o implementar dependendo da complexidade do próprio e do esquema da base de dados. O trabalho do query planner é selecionar um algoritmo de entre todas as hipóteses que dê uma resposta com o mínimo de I/O e CPU.

- Análise do WHERE

A cláusula WHERE de uma query é repartida em "termos" onde cada termo é separado dos outros pelo operador AND. Se a cláusula for composta por restrições separadas pelo operador OR então a clausula e considerada num único termo onde é aplicada a optimização do OR.

Todos os termos do WHERE são analisados para ver se é possível satisfaze-los usando índices. Para um índice ser usado o termo tem de ser de uma das seguintes formas:

```
column = expression
column > expression
column >= expression
column < expression
column <= expression
expression = column
expression > column
expression >= column
expression < column
expression <= column
column IN (expression-list)
column IN (subquery)
column IS NULL
```

Se um índice for criado usando um statement do tipo:

```
CREATE INDEX idx_ex1 ON ex1(a,b,c,d,e,...,y,z);
```

Então o índice pode ser usado se as colunas iniciais do índice aparecerem nos termos do WHERE. As colunas iniciais do índice têm de ser usadas com os operadores =, IN ou IS NULL, sendo que a última coluna pode usar desigualdades.

Não é necessário todas as colunas de um índice aparecerem no WHERE para que se possa usar o mesmo mas não podem haver falhas entre as colunas que são usadas. Isto é, usando o exemplo acima, se não houver uma restrição na coluna c no WHERE então apenas os termos que restringem as colunas a e b podem ser usados com o índice.

- Exemplos de uso de índices (todos os exemplos usam o índice acima)

```
... WHERE a=5 AND b IN (1,2,3) AND c IS NULL AND d='hello'
```

As primeiras quatro colunas do índice podem ser usadas

```
... WHERE a=5 AND b IN (1,2,3) AND c>12 AND d='hello'
```

Apenas as colunas a, b e c do índice podem ser usadas pois c está restringido apenas por desigualdades

```
... WHERE a=5 AND b IN (1,2,3) AND d='hello'
```

Apenas as colunas a e b do índice podem ser usadas, pois a coluna c não está restringida e não pode haver falhas no conjunto de colunas a usar

```
... WHERE b IN (1,2,3) AND c NOT NULL AND d='hello'
```

O índice não pode ser usado porque a coluna mais a esquerda (a) não está restringida no WHERE. A query iria resultar num full table scan.

```
... WHERE a=5 OR b IN (1,2,3) OR c NOT NULL OR d='hello'
```

O índice não pode ser usado porque os termos estão ligados por OR em vez de AND. Se fossem adicionados 3 novos índices onde a primeira coluna fosse respetivamente b, c e d então a optimização do OR poderia ser aplicada

- Optimização do BETWEEN

Se um termo do WHERE tiver a seguinte forma:

```
expr1 BETWEEN expr2 AND expr3
```

Então 2 termos "virtuais" são adicionados:

```
expr1 >= expr2 AND expr1 <= expr3
```

Estes termos são usados apenas para analise e não causam a geração de nenhum código. Se ambos os termos forem usados como restrições num índice então o termo BETWEEN original é omitido e o teste correspondente não é realizado. Sendo assim se os termos do BETWEEN forem usados como restrições de um índice então nenhum teste é realizado no termo original. Por outro lado, termos virtuais nunca causam testes no input, logo se o BETWEEN não é usado como restrição de um índice e é usado para teste de input então a expr1 é avaliada apenas uma vez.

- Optimizações OR

Restrições de um WHERE que estão ligadas com OR em vez de AND podem ser tratadas de duas maneiras. Se um termo consiste de vários subtermos que contêm uma coluna em comum e estão separados por OR, por exemplo:

```
column = expr1 OR column = expr2 OR column = expr3 OR ...
```

Então esse termo é reescrito da seguinte maneira:

```
column IN (expr1,expr2,expr3,...)
```

O termo reescrito pode então ser usado para restringir um índice usando as regras normais para os operadores IN.

Se a conversão de OR para IN não funcionar, a segunda optimização é usada. Suponha-se que o OR consiste de vários subtermos, por exemplo:

```
expr1 OR expr2 OR expr3
```

Cada subtermo é analisado como se fosse uma cláusula WHERE de modo a verificar se o subtermo é indexável por si próprio. Se todos os termos forem indexáveis separadamente então o OR pode ser codificado de modo a que um índice diferente seja usado para avaliar cada termo.

Note-se que na maioria dos casos, o SQLite usa um único índice para cada tabela no FROM de uma query. A segunda optimização aqui descrita é a exceção à regra.

- Optimização Skip-Scan

Regra geral os índices apenas são uteis se houver restrições do WHERE nas colunas da esquerda do índice. Contudo, em alguns casos, o SQLite é capaz de usar um índice mesmo que as primeiras colunas sejam omitidas do WHERE.

Considere-se a tabela seguinte:

```
CREATE TABLE people(
   name TEXT PRIMARY KEY, role TEXT NOT NULL,
   height INT NOT NULL, -- in cm
   CHECK( role IN ('student', 'teacher') )
);
CREATE INDEX people_idx1 ON people (role, height);
```

A tabela pessoas tem uma entry para cada pessoa. Existe um índice para role e height, sendo que a primeira coluna do mesmo não é muito seletiva tendo apenas dois valores possíveis.

Considere-se agora a seguinte query:

```
SELECT name FROM people WHERE height>=180;
```

Como a primeira coluna do índice não aparece no WHERE da query então não seria possível usar o índice. Mas o SQLite consegue usar o índice. Conceptualmente o SQLite usa o índice como se a query fosse:

```
SELECT name FROM people

  WHERE role IN (SELECT DISTINCT role FROM people)

    AND height>=180;
```

Ou:

```
SELECT name FROM people WHERE role='teacher' AND height>=180

UNION ALL

SELECT name FROM people WHERE role='student' AND height>=180;
```

O SQLite não transforma verdadeiramente a query. O verdadeiro plano de avaliação da query é:

O SQLite localiza o primeiro valor possível para "role", rebobinando o índice para o início, e guarda-o numa variável interna "$role". Posteriormente corre uma query do tipo: "SELECT name FROM people WHERE role=$role AND height>=180". Esta query já pode ser avaliada usando o índice. Quando essa terminar o SQLite volta a usar o índice para localizar o próximo valor da coluna "role" e usa o mesmo para repetir a query com o novo valor até que todos os valores possíveis do "role" tenham sido examinados

A única maneira de o SQLite saber que as primeiras colunas do índice têm muitos duplicados (situação onde usa o skip-scan) é se o comando ANALYZE tiver sido usado na base de dados. Sem o comando ANALYZE, o SQLite usa um valor por defeito de 10 para o número medio de duplicados por coluna no índice mas o skip-scan apenas é útil a partir dos 18 ou mais duplicados. Logo, o skip-scan nunca será usado se a base de dados não tiver sido analisada

- Joins

Considere-se as seguintes queries:

```
SELECT * FROM tab1 LEFT JOIN tab2 ON tab1.x=tab2.y;

SELECT * FROM tab1 LEFT JOIN tab2 WHERE tab1.x=tab2.y;
```

Para um LEFT OUTER JOIN estas duas queries não seriam equivalentes, mas para um INNER JOIN sim. Quando se avalia as cláusulas ON e USING de um OUTER JOIN as restrições das cláusulas não se aplicam se a tabela da direita do join estiver numa linha null, mas aplicam se for um WHERE.

- Ordem das tabelas num join

De momento o SQLite implementa todos os joins como nested loops.

A ordem por defeito dos loops num join é que a tabela da esquerda forma o loop externo e a tabela da direita o interno. Contudo, o SQLite consegue mudar a ordem se tal ajudar a selecionar índices melhores

Inner joins podem ser reordenados a vontade, contudo os left outer joins não são comutativos nem associativos não podendo ser reordenados.

A reordenação é automática e geralmente funciona bem o suficiente para que os programadores não tenham de pensar nisso, especialmente se o ANALYZE tiver sido usado. Mas ocasionalmente algumas pistas têm de ser dadas pelo programados. Considere-se o seguinte esquema:

```
CREATE TABLE node(
      id INTEGER PRIMARY KEY,
      name TEXT
);
CREATE INDEX node_idx ON node(name);
CREATE TABLE edge(
      orig INTEGER REFERENCES node,
      dest INTEGER REFERENCES node,
      PRIMARY KEY(orig, dest)
);
CREATE INDEX edge_idx ON edge(dest,orig);
```

Considere-se agora a seguinte query:

```
SELECT *
    FROM edge AS e,
           node AS n1,
           node AS n2
 WHERE n1.name = 'alice'
     AND n2.name = 'bob'
     AND e.orig = n1.id
     AND e.dest = n2.id;
```

O que esta query pede é toda a informação sobre as arestas que vão de nós com nome "alice" para nós "bob". O optimizador tem basicamente duas opções sobre como implementar esta query:

Opção 1:

```
foreach n1 where n1.name='alice' do:
    foreach n2 where n2.name='bob' do:
      foreach e where e.orig=n1.id and e.dest=n2.id
          return n1.*, n2.*, e.*
      end
    end
end
```

Opção 2:

```
foreach n1 where n1.name='alice' do:
    foreach e where e.orig=n1.id do:
      foreach n2 where n2.id=e.dest and n2.name='bob' do:
          return n1.*, n2.*, e.*
      end
    end
end
```

Os mesmos índices são usados para acelerar cada loop em ambas as opções. A única diferença entre os dois planos é a ordem dos loops.

A escolha da melhor opção vai depender do tipo de dados de cada nó.

Seja o numero de nós alice M e o numero de nós bob N. Considere-se dois cenários. No primeiro, M e N são ambos 2 mas há milhares de arestas em cada nó. Neste caso a opção 1 seria melhor. Com a opção 1, o loop interno verifica a existência de uma aresta entre um par de nós e devolve o resultado encontrado. Mas como apenas há dois nós com alice e bob, o loop apenas tem de correr 4 vezes. A opção 2 seria muito mais demora aqui. O loop externo apenas executa 2 vezes, mas devido ao numero de arestas o loop do meio teria de iterar milhares de vezes.

Considere-se agora M e N são ambos 3500. Suponha-se que cada um dos nós está ligado por apenas uma ou duas arestas. Neste caso a opção 2 é preferível. Com a opção 2 o loop externo ainda tem de correr 3500 vezes, mas o loop do meio apenas corre uma ou duas vezes para cada loop externo e o interno apenas uma vez para cada loop do meio. Por isso o numero total de iterações do loop interno é cerca de 7000. Com a opção 1, quer o loop interno que o do meio têm de correr 3500 vezes, resultando em 12 milhões de iterações do loop do meio.

- Controlo Manual de query plans usando SQLITE_STAT

O SQLite fornece aos programadores avançados a capacidade de controlar o plano escolhido pelo optimizador. Uma forma de o fazer é "falsificar" os resultados do ANALYZE nas tabelas sqlite_stat1, sqlite_stat3, e/ou sqlite_stat4. Essa abordagem não é recomendada.

- Controlo Manual de query plans usando CROSS JOIN

Programadores podem forcar o SQLite a usar uma ordem especifica de loops para um join usando o operador CROSS JOIN. Apesar de, em teoria, o CROSS JOIN ser comutativo, o SQLite escolhe nunca mudar a ordem das tabelas de um CROSS JOIN. Sendo assim, a tabela da esquerda de um CROSS JOIN será sempre o loop externo e a da direita o interno.

Na seguinte query, o optimizador pode reordenar as tabelas como quiser:

```
SELECT *
    FROM node AS n1,
           edge AS e,
           node AS n2
 WHERE n1.name = 'alice'
     AND n2.name = 'bob'
     AND e.orig = n1.id
     AND e.dest = n2.id;
```

Mas nesta, ao substitui a "," por "CROSS JOIN", a ordem das tabelas tem de ser a indicada:

```
SELECT *
    FROM node AS n1 CROSS JOIN
           edge AS e CROSS JOIN
           node AS n2
 WHERE n1.name = 'alice'
     AND n2.name = 'bob'
     AND e.orig = n1.id
     AND e.dest = n2.id;
```

- Escolha entre vários indices

Cada tabela na cláusula FROM de uma query pode usar no máximo um índice e o SQLite esforça-se para usar pelo menos um índice em cada tabela. Por vezes, dois ou mais índices podem ser candidatos para ser usados numa única tabela. Por exemplo:

```
CREATE TABLE ex2(x,y,z);
 CREATE INDEX ex2i1 ON ex2(x);
 CREATE INDEX ex2i2 ON ex2(y);
 SELECT z FROM ex2 WHERE x=5 AND y=6;
```

Para o SELECT acima, o optimizador pode usar o índice ex2i1 para ver as linhas da tabela ex2 que contêm x=5 e depois testar cada linha contra o termo y=6, ou pode usar o índice ex2i2 para ver as linhas da tabela que contêm y=6 e testar cada uma contra o termo x=5.

Quando confrontado com a escolha entre dois ou mais índices, o SQLite tenta estimar o trabalho total necessário para realizar a query usando cada opção, escolhendo a com o menor valor.

- Desqualificação de termos do WHERE usando "+" unário

Os termos de uma cláusula WHERE podem ser manualmente desqualificados do uso com índices prefixando-os com um + unário. O + unário não gera nenhum código mas previne o termo de restringir um índice. Sendo assim, no exemplo acima, se a query fosse reescrita:

```
SELECT z FROM ex2 WHERE +x=5 AND y=6;
```

O operador + na coluna x iria prevenir que o termo restringisse um índice forçando o uso do índice ex2i2.

- Range Queries

Considere-se um cenário diferente:

```
CREATE TABLE ex2(x,y,z);

CREATE INDEX ex2i1 ON ex2(x);

CREATE INDEX ex2i2 ON ex2(y);

SELECT z FROM ex2 WHERE x BETWEEN 1 AND 100 AND y BETWEEN 1
AND 100;
```

Suponha-se que a coluna X conte valores distribuídos entre 0 e 1000000 e que a coluna Y conte valores entre 0 e 1000. Nesse cenário a restrição de range na coluna x reduz o espaço de procura por um factor de 10000 onde a restrição da coluna y reduz por um factor de apenas 10. Logo o índice ex2i1 é preferível

O SQLite apenas irá fazer esta determinação se for compilado com SQLITE_ENABLE_STAT3 ou SQLITE_ENABLE_STAT4. Estas opções fazem com que o comando ANALYZE recolha um histograma dos conteúdos das colunas para as tabelas sqlite_stat3 ou sqlite_stat4 e use o histograma para fazer melhores suposições sobre a melhor query a usar para restrições de range como a acima.

Os dados do histograma só são uteis se o lado direito da restrição for uma simples constante ou parâmetro e não uma expressão. Outra limitação dos dados do histograma é que apenas se aplicam à primeira coluna de um índice. Considere-se este cenário:

```
CREATE TABLE ex3(w,x,y,z);

CREATE INDEX ex3i1 ON ex2(w, x);

CREATE INDEX ex3i2 ON ex2(w, y);

SELECT z FROM ex3 WHERE w=5 AND x BETWEEN 1 AND 100 AND y
BETWEEN 1 AND 100;
```

Aqui as desigualdades estão nas colunas x e y que não são as primeiras colunas nos índices. Sendo assim, os dados do histograma são inúteis para ajudar a escolher entre as restrições das colunas x e y.

- Nivelamento de subquery

Quando existe uma subquery dentro do FROM de um SELECT, o comportamento mais simples é avaliar a mesma para uma tabela transiente, e posteriormente correr o SELECT exterior contra a tabela. Mas esse plano pode não ser óptimo já que a tabela nova não vai ter nenhum índice e a tabela exterior será forçada a fazer um scan da tabela inteira.

Para ultrapassar este problema o SQLite tenta nivelar as subqueries. Isto envolve inserir a clausula FROM da subquery no FROM da query exterior e a reescrita das expressões na query exterior que referem o resultado da subquery. Por exemplo:

```
SELECT a FROM (SELECT x+y AS a FROM t1 WHERE z<100) WHERE a>5
```

Seria reescrita para:

```
SELECT x+y AS a FROM t1 WHERE z<100 AND a>5
```

Há uma longa lista de condições que deve todas ser cumpridas para que se possa realizar o nivelamento de queries.

- Índices Automáticos

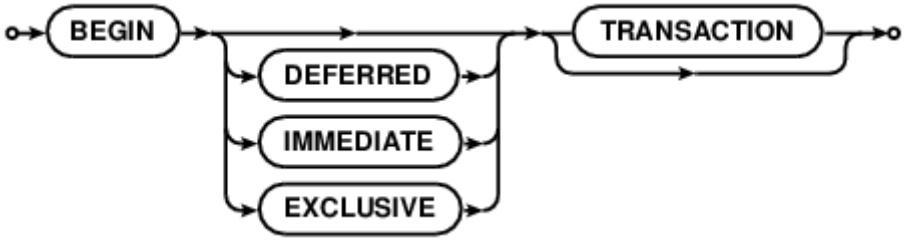
Quando não há índices disponíveis o SQLite pode criar um índice automático que apenas existe durante um único statement SQL. Como o custo da construção de um índice automático é O(N*log N) (onde N é o numero de entries na tabela) e o custo de um full table scan é apenas O(N), um índice automático só será criado se o SQLite esperar que o lookup será executado mais que log N vezes durante o curso do statement SQL

Gestão de transações e controlo de concorrência

Uma base de dados que permita boas transações é aquela em que todas as operações sob queries respeitam as normas ACID. O SQLite implementa transações do tipo serializable que respeitam estas normas, mesmo que uma transação seja interrompida quer por *crash* de um programa, de um *crash* do sistema ou por falha energética.

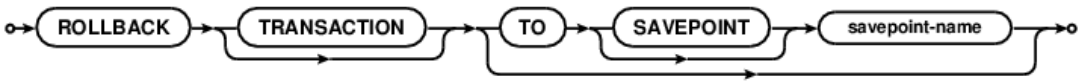
Em SQLite temos, representadas nos seguintes esquemas, as situações no que toca a transações:

Begin **Statment:**



Commit Statment:

Rollback Statment:



Não podem ser efetuadas alterações à base de dados exceto entre transações. Qualquer comando que altere a base de dados irá começar automaticamente uma transação se nenhuma outra estiver a ser executada. Estas ficam em estado committed assim que acaba a última linha da query.

As transações podem ser começadas manualmente através do comando BEGIN. Tais transações persistem normalmente, até um comando COMMIT ou ROLLBACK. Mas uma transação poderá fazer ROLLBACK se a base de dados for fechada ou se ocorrer um erro que lance o algoritmo de resolução do mesmo.

Transações criadas com BEGIN…COMMIT não são do tipo *nested*, para transações deste tipo, utiliza-se as transações SAVEPOIT e RELEASE to SAVEPOIT. Uma tentativa de invocar o comando BEGIN entre transações resulta em erro, não querendo saber se esta começou com um BEGIN ou com um SAVEPOINT.

Tipos de Transações:

O SQLite tem três tipos diferentes de transações em diferentes estados de *lock*. Estas podem ser do tipo ***deferred***, ***immediate***, ou ***exclusive***, sendo este especificado no comando BEGIN:

```
begin [deferred | immediate | exclusive] transaction;
```

Deferred – Não adquire nenhum *lock* até que tenha de o fazer. Assim com este de tipo de transação a instrução BEGIN não faz nada em si, pois dá origem a um estado bloqueante, isto por defeito. Se simplesmente se emitir um BEGIN, então a transação é adiada (***derrefed***) e, portanto, fica-se em estado desbloqueado. Múltiplas sessões podem começar simultaneamente transações deste tipo ao mesmo tempo sem se criar qualquer bloqueio (***lock***). Neste caso q primeira instrução de leitura à base de dados adquire um *lock* compartilhado, e similarmente a primeira operação de escrita tenta adquirir um lock reservado.

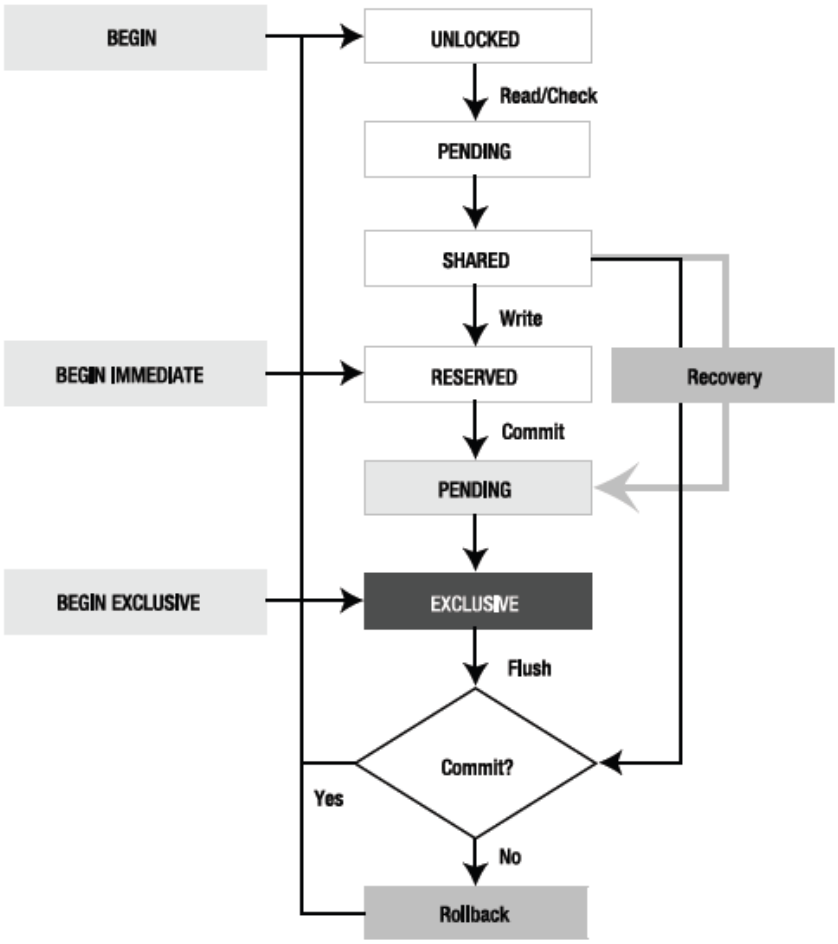
Immediate – Tenta obter um *lock* reservado tão cedo quanto o comando BEGIN é executado. Se for bem-sucedido, esta transação garante que nenhuma outra sessão consiga escrever na base de dados. É sabido que outras sessões podem continuar a ler dados da base de dados, mas este tipo de *lock* proíbe novas sessões efetuar a operação de leitura. Outra consequência é que nenhuma outra sessão será capaz de emitir com sucesso um comando *Immediate* ou *Exclusive*. O SQLite devolve o erro SQLITE_BUSY. Isto significa que estão outras sessões de leitura ativas. Uma vez que estas terminem, poderá ser feito o *commit* da transação.

Exclusive – O funcionamento é semelhante ao do **Immediate**, mas esta garante que nenhuma outra sessão está ativa na base de dados e que podemos efetuar operações de leitura e escrita livremente. Neste caso, apenas uma das sessões terá conseguido entrar no **Immediate**, enquanto a outra terá de aguardar. Esta pode continuar a tentar com a garantia que, eventualmente, entrará numa transação **Immediate** ou **Exclusive**. Se este método for utilizado por todas as sessões que queiram fazer operações de escrita à base de dados então, estará a ser fornecido um mecanismo de sincronização que previne *deadlocks*.

Estados de um Lock:

Na maior parte dos casos, a duração de um *lock* oculta a duração da transação. Embora ambas nem sempre comecem juntas, estas terminam sempre ao mesmo tempo. Um *lock* nunca é libertado até que a sua transação conclua ou, no pior caso, o programa rebente e a transação não se conclua.

O diagrama abaixo mostra todos os estados possíveis de um lock, tal como os todos os caminhos que este pode tomar no ciclo de vida de uma transação:



UNLOCKED	Não há locks na Base de Dados. Esta pode ser lida ou escrita. Todos os dados armazenados internamente em cache são considerados suspeitos e sujeitos a verificação contra o ficheiro da Base de Dados antes de serem usados. Outros processos poderão ler ou escrever de acordo com o que o seu *lock* permita. Este é o estado por definição.
SHARED	A Base de dados pode ser lida mas não escrita. Qualquer número de processos pode ter *SHARED* locks ao mesmo tempo, por isso pode haver leituras simultâneas. Mas nenhuma outra *thread* ou processo pode escrever na Base de Dados enquanto um ou mais *SHARED locks* estiverem ativos.
RESERVED	Significa que o processo está a pensar em escrever no ficheiro da Base de Dados, mas que de momento está apenas a lê-lo. Apenas um *RESERVED* pode estar ativo de cada vez, embora múltiplos *SHARED locks* possam coexistir com um *RESERVED lock.*
PENDING	Aqui o processo que tem o *lock* quer escrever na Base de Dados o mais rápido possível e está apenas à espera que todos os processos com *SHARED* terminem para que este consiga obter um *EXCLUSIVE lock*. Não são permitidos novos *SHARED locks* contra a base de dados se um *PENDING lock* estiver ativo, embora os *SHARED locks* existentes tenham permissão para continuar.
EXCLUSIVE	É necessário para se poder realizar a operação de escrita na Base de Dados. Apenas um *EXCLUSIVE lock* é permitido no ficheiro e nenhum outro *lock* de qualquer espécie pode coexistir com um *EXCLUSIVE lock.* De maneira a maximizar a concorrência, o SQLite trabalha de forma a minimizar a quantidade de tempo que um *EXCLUSIVE locks* é mantido.

Exemplo de deadlock:

Consideremos o seguinte cenário onde duas sessões, A e B estão a operar sob a mesma base de dados ao mesmo tempo. A ordem dos comandos é a apresentada, primeiro A, as duas seguintes são de B...

Session A	Session B
sqlite> begin;	
	sqlite> begin;
	sqlite> insert into foo values ('x');
sqlite> select * from foo;	
	sqlite> commit;
	SQL error: database is locked
sqlite> insert into foo values ('x');	
SQL error: database is locked	

Ambas as sessões acabam em *deadlock*. A sessão B foi a primeira a tentar escrever na base de dados e assim sendo tem um *lock* pendente. A sessão A tenta escrever mas falha quando o seu INSERT tenta promover o lock partilhado a reservado. A titulo de exemplo imaginando que A espera até que se possa escrever e B também. Neste caso, tudo o resto ficaria bloqueado, se uma terceira sessão fosse aberta, esta nem conseguiria ler da base de dados. A razão para tal acontecer é B ter um lock pendente, o qual previne as outras sessões de adquirirem *locks* partilhados. Por isso não só A e B estão bloqueados como bloquearam tudo o resto que está fora da base de dados. A solução, claro está, resulta em escolher o escalonamento correto das transações.

Suporte para bases de dados distribuídas

A versão Cliente/Servidor do SQLite ainda está restringida pela capacidade de processamento de um único sistema servidor. A única maneira de superar isto é com o uso de um cluster ou cloud de sistemas servidor. A abordagem mais comum é o uso de replicação master/slave ignorando ACID correctness.

Usando um Group Communication System para transmitir as informações de atualização a todos os nós participantes parece ser uma arquitectura simples de alcançar performance escalável e ACID correctness ao mesmo tempo. Alguns trabalhos em Postgres mostrarem a abordagem escalada para 15 nós.

Um GCS bem projectado é cerca de 20000 linhas de código, cerca de dois terços do tamanho do código de toda a biblioteca do SQLite. Usar esta abordagem tem a promessa de ser um design radical e rápido para um servidor de bases de dados distribuídas.

Outras características do sistema estudado

Zero configuration e Serverless

O SQLite distingue-se por ser uma database engine sem configurações necessárias, e sem servidor. Ao contrário da maioria das engines, em que um processo servidor está em execução e serve de intermediário, respondendo aos pedidos de outros processos, com o SQLite os processos acedem directamente aos ficheiros de bases de dados para operações de leitura e escrita. Qualquer programa com acesso ao disco consegue aceder a uma base de dados do SQLite.

Um único ficheiro, um único formato de ficheiro para todas as plataformas

Uma base de dados SQLite está inteiramente contida num único ficheiro, e como foi referido, qualquer programa com acesso ao disco consegue ler e escrever sobre a base de dados. Por um lado, os dados estão menos seguros, mas por outro, são mais facilmente acedidos. Este ficheiro é facilmente copiado para uma pen drive, ou partilhado por email, e tem a grande vantagem de poder ser lido em qualquer plataforma, de diferentes arquitecturas, porque o formato de ficheiro utilizado é sempre o mesmo.

Extremamente compacto

As bibliotecas são extremamente compactas. Ao obter as bibliotecas do SQLite dos repositórios Ubuntu, numa máquina x64, o espaço total ocupado em disco não ultrapassou 1 MB (cerca de 700KB), podendo até ocupar menos espaço.

Manifest Typing

O SQLite não utiliza static typing, ou seja, não força que um valor de uma coluna seja do mesmo tipo declarado para essa coluna. Existem algumas excepções, por exemplo, colunas declaradas chaves primárias Integer só podem conter valores Integer. Esta tipificação traz algumas vantagens, facilitando a combinação entre o SQLite e linguagens com tipificação dinâmica (Python, Ruby, Objective-C).

ODBC e JDBC

Estão disponíveis separadamente drivers ODBC e JDBC para o SQLite. No endereço http://www.ch-werner.de/sqliteodbc/ é possível transferir uma driver ODBC, para Windows, Linux e MacOS. No endereço https://bitbucket.org/xerial/sqlite-jdbc encontra-se a biblioteca sqlite-jdbc.

Suporte para diferentes linguagens de programação

Inicialmente, o SQLite foi desenvolvido como uma extensão para a linguagem TCL. Os bindings para TCL estão incluídos nas builds do SQLite. Também estão disponíveis bindings para muitas outras linguagens, embora separadamente. Algumas das linguagens suportadas são o Python, PHP, Ruby, Objective-C...

Suporte Web

Naturalmente, existe suporte Web no SQLite, uma vez que, como já foi mencionado anteriormente, qualquer aplicação com acesso ao disco consegue efectuar operações de escrita e leitura sobre a base de dados. Embora seja uma database engine direccionada para bases de dados mais pequenas, não sendo apropriada para grandes volumes de informação, com elevada concorrência, o SQLite tem um bom desempenho para websites com tráfego moderado.

Comparação com o Oracle 11g

É importante referir que dado o SQLite ser um sistema grátis ao contrário do Oracle que é pago, é normal que este tenha menos recursos face ao Oracle.

Ambos os sistemas se baseiam no modelo relacional, no entanto embora o Oracle tem um tipo de sistema estático e dinâmico com interface GUI e SQL, enquanto o SQLite tem apenas um sistema estático e uma interface SQL.

Sistemas Operativos:
Relativamente a sistemas operativos, o SQLite não correm em z/OS ao contrário do Oracle, no entanto corre em sistemas BSD, Symbian e AmigaOS onde o Oracle não opera. Ambos conseguem operar em sistemas Windows, Mac OS X, UNIX e Linux.

Especificações:

Oracle	SQLite
• Controlo de Acesso:	
✔ Brute-force Protection	✘ Brute-force Protection
✔ Enterprise Directory Compatibility	✘ Enterprise Directory Compatibility
✔ Native Network Encryption	✘ Native Network Encryption
✔ Password Complexity Rules	✘ Password Complexity Rules
✘ Patch Access	✔ Patch Access
✘ Run Unprivileged	✔ Run Unprivileged
✔ Security Certification	✘ Security Certification
✔ Audit	✔ Audit
✔ Resource Limit	✔ Resource Limit
• Índices:	
✔ Bitmap	✘ Bitmap
✔ Expression	✘ Expression
✔ Hash	✘ Hash
✔ Partial	✘ Partial
✔ Full-text	✔ Full-text
✔ R-/R+ Tree	✔ R-/R+ Tree
✔ Reverse	✔ Reverse

- Particionamento:

✔ Composite (Range + Hash)	✘ Composite (Range + Hash)
✔ Hash	✘ Hash
✔ List	✘ List
✘ None	✔ None
✔ Range	✘ Range

- Ouros Objetos:

✔ Cursor	✘ Cursor
✔ Data Domain	✘ Data Domain
✔ Function	✘ Function
✔ Procedure	✘ Procedure
✔ External Routine	✔ External Routine
✔ Trigger	✔ Trigger

- Tabelas e Vistas:

✔ Materialized Views	✘ Materialized Views
✔ Temporary Table	✔ Temporary Table

Capacidades da Base de Dados:

✔ Common Table Expressions	✘ Common Table Expressions
✔ Merge Joins	✘ Merge Joins
✔ Outer Joins	✘ Outer Joins
✔ Parallel Query	✘ Parallel Query
✔ Windowing Functions	✘ Windowing Functions
✔ Blobs and Clobs	✔ Blobs and Clobs
✔ Except	✔ Except
✔ Inner Joins	✔ Inner Joins
✔ Inner Selects	✔ Inner Selects
✔ Intersect	✔ Intersect
✔ Union	✔ Union

Tipos de Dados:

Em Oracle as variáveis de tipo **inteiro** são dadas como NUMBER, os de **vírgula flutuante** como BINARY_DOUBLE ou BINARY_FLOAT, os **decimais** como NUMERIC e **strings** como CHAR, VARCHAR, NCHAR e NVARCHAR.
Em SQLite o cenário é diferente sendo os **inteiros** dados como INTEGER8 (64 bits), os **não inteiros** como REAL e as **strings** como TEXT

Tamanhos:

No SQLite o tamanho máximo de um CLOB (Character Large Object) é de aproximadamente 1 GB e o tamanho máximo para uma base de dados é de relativamente 32 TB**.**

Comandos:

Além de SQL, dispõe de mais comandos particulares do SQLite

- .help - lista comandos disponíveis (além do SQL)
- .read nomeFicheiro - executa script SQL
- .schema - lista esquemas das tabelas
- .quit - sair do interpretador
- .tables - lista as tabelas
- .separator sep - especifica o separador dos campos
- .import nomeFicheiro tabela - importa o ficheiro para a tabela assumindo os campos separados pelo separador definido

Bibliografia

http://en.wikipedia.org/wiki/SQLite

http://www.sqlite.org

http://www.sqliteconcepts.org/DCS_index.html

http://www.tutorialspoint.com/sqlite/

http://database-management-systems.findthebest.com/compare/36-53/Oracle-vs-SQLite

[Slides FCT-UNL] Informática para Ciências e Engenharias 2013/14 (Teórica 10)

[Allen, Gran; Owens, Mike] The Definitive Guide to SQLite – 2nd Edition

[Haldar, Sibsankar] SQLite Database System: Design and Implementation