D-036

SQLite へのハッシュ結合の導入と評価
An implementation of the hash join on SQLite

片山 大河† 嶋村 誠† 山地 圭† 金松 基孝†
Taiga Katayama Makoto Shimamura Kei Yamaji Mototaka Kanematsu

1. まえがき

SQLite[1]はリレーショナルデータベース管理システム（RDBMS）のひとつで，OSS として公開されており組込み機器をはじめとする多くの製品に利用されている．例えば Android で標準サポートされているなど，近年ますますその利用シーンが増加している．

RDBMS への代表的な操作としてテーブルの結合がある．テーブル結合とは，2 つ以上のテーブルに対してそれぞれの特定のカラムの値を基にしてひとつのテーブルにまとめる操作である．この操作はアプリケーションの性能に直結する重要な操作のひとつであり，この処理の高速化が求められている．

結合アルゴリズムには，ネステッドループ結合，ハッシュ結合，ソートマージ結合などいくつか種類があり，データ量や検索条件によって向き不向きがある．どの結合アルゴリズムを使用するかは，統計情報やユーザに与えられるヒントに基づき決定される．ハッシュ結合とは，テーブル結合のうちよく使用される等価結合を高速に行える結合アルゴリズムで，カラム値の重複が少ないケースで高速に結合できる．等価結合とは複数のテーブル間でカラムの値が一致する行を取り出してひとつの表にまとめる処理である．

しかし，SQLite は Btree インデックスを使ったネステッドループ結合しかできない．そこで，本論文ではハッシュ結合を SQLite に導入することで等価結合の高速化を行う．

2. 設計方針と SQLite の結合処理

SQLite は，索引がないカラムに対する結合を行うとき実行時に一時的な索引を作成して高速化を図る仕組みがある（自動索引と呼ぶ）．一方のテーブルに対して索引を作成し，もう一方のテーブルの結合対象カラムの値（探索キーと呼ぶ）を探索する（図 1破線）．

今回は自動索引の代替としてハッシュ表を作成するという方針でハッシュ結合を実現した．図 1実線にハッシュ結合の流れを示す．まず入力テーブル（ビルド入力と呼ぶ）に対してハッシュ値を計算し，ハッシュ表に行番号を登録する．ハッシュ表の作成が完了したら探索を行う．探索キー（ハッシュ表を探索する際はこれをプローブ入力と呼ぶ）のハッシュ値を計算し，ハッシュ表から行番号（id）を取り出す．そしてその行番号をもとにデータにアクセスをする．

これを実現するために，SQLite のプランナを改造しハッシュ結合用の実行計画を生成するようにする．プランナとは，実行エンジンが行う処理手順である実行計画を作成するモジュールである．SQLite の実行計画はオペコ

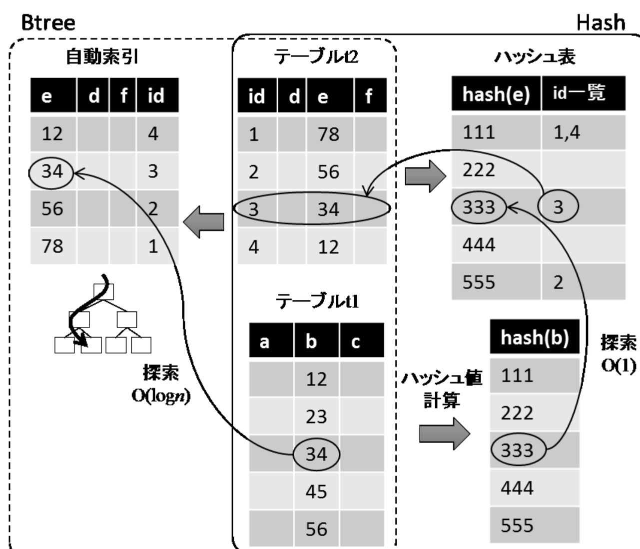

図1 結合イメージ

ードの組み合わせで構成される．オペコードとは実行エンジンが実行する処理の最小単位である．実行計画は上から順番にひとつずつオペコードが実行される．オペコードによっては条件次第でジャンプするものがあり，図 2中の矢印はそのジャンプ先を示している．

以下では SQLite のプランナが作成する実行計画について説明する．図 2(左)に自動索引による結合を行う典型的な実行計画の抜粋を示す．処理内容はカーソルの初期化，索引の作成，索引からの探索で構成されている．図 2（右）のハッシュ結合については3章で述べる．

2.1. カーソルの初期化

カーソルとはテーブルや索引などに格納されているデータにアクセスするためのデータ構造で，特定の一行を指すポインタのようなものである．

まず，オペコード *OpenRead*（以降，オペコードは斜体の英単語で表記する）で結合に使用する 2 つのテーブルに対する参照用カーソルを作成し，先頭行にカーソル合わせる．一方が索引構築のための入力，もう一方が探索キーのカーソルである．これらはハッシュ結合ではそれぞれビルド入力とプローブ入力のカーソルになるものである．

2.2. 索引の作成

図 2左上のように，*OpenAutoindex* により自動索引格納場所の用意と索引用のカーソルの作成を行う．次に入力テーブルのカーソルを使用して *Column* でカラム値を取得し *Rowid* で行番号を取得する．そして取得した値を使用して *MakeRecord* で一行分のエントリを作成した後

†株式会社東芝 インダストリアル ICT ソリューション社

Copyright © 2015 by Information Processing Society of Japan and
The Institute of Electronics, Information and Communication Engineers
All rights reserved.

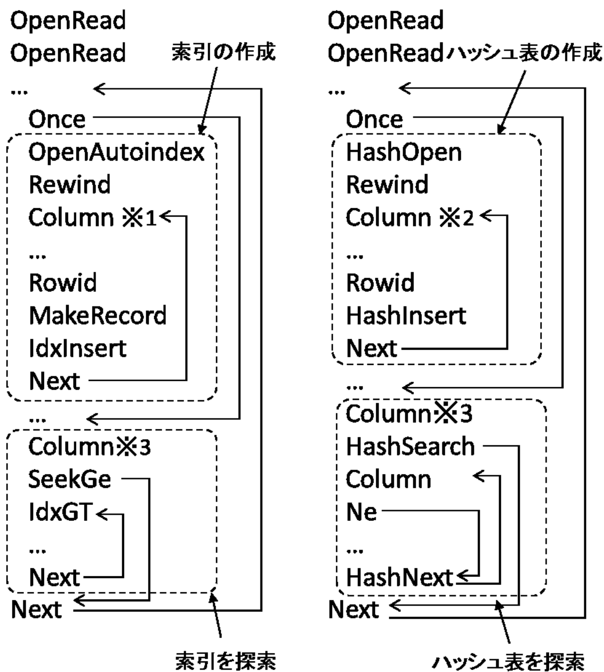

図2 結合処理の実行計画例 (左)自動索引 (右)ハッシュ

IdxInsert で索引に登録する．この登録処理は *Next* によりテーブルの行数だけ繰り返す．

索引の作成において SQLite の仕様として特筆することが 2 つある．1 つ目は索引のエントリに含むカラムは結合キーだけでないことである．2 つ目は異なる型のデータの混在を許すことである．

2.2.1. 索引作成対象のカラム

索引作成対象のテーブルについて，アクセスする全てのカラム値からエントリを作成する．例えば，SQL 文

SELECT x.a, t2.d FROM x INNER JOIN y ON x.b = y.e WHERE x.c = ‘A’ AND y.f = ‘B’;

においてテーブル y の索引を作成する場合，結合キーである y.e だけでなく，射影対象の y.d および選択条件で使用する y.f も索引のエントリに含まれる．そのため，結合条件に一致した行のカラム値を取得する際は，索引構築時の入力カーソルではなく，索引用のカーソルを使用するように設計されている．図 2中の※1 の *Column* がエントリに含むカラムの数だけ現れることになる．データのコピーが多く発生して無駄であるため，ハッシュ表の生成では結合キーのカラムのみをハッシュ表のエントリに使用するようにする．詳細は3.2で述べる．

2.2.2. 異なる型のデータの混在

SQLite は同一カラムでも異なる型のデータを格納可能である．例えば，次のようなことが可能である．

INSERT INTO x(a) VALUES(123);
INSERT INTO x(a) VALUES(‘123’);

さらに，型が異なるデータも等価とすることである．例えば，次のように整数の 123 が格納されたテーブル t1 と文字列の’123’が格納されたテーブル t2 がある．

INSERT INTO x(b) VALUES(123);
INSERT INTO y(f) VALUES(‘123’);

この 2 つのテーブルを結合すると，次の SQL 文は 1 件ヒットする．

SELECT * FROM t1 INNER JOIN t2 ON t1.b = t2.f;

Type Affinity という概念[2]があり，そのカラムをどの型で扱うかあるいはその比較演算をどの型で行うかが定義されて，データはなるべくその型に合うように変換がされる．この仕組みをハッシュ結合でも機能させるために，ハッシュ表の生成前に型を合わせる処理が必要になる．詳細は3.3で述べる．

2.3. 索引からの探索と結合演算

索引のエントリは整列されている．図 2左下のように，まず *SeekGe* により探索キーと一致する最初のエントリを探し出す．*IdxGT* で索引キーに一致するエントリかを判定することで結合結果が 1 行得られる．

この *IdxGT* は，*Next* のループにより索引用カーソルを 1 つずつずらして，一致しなくなるまで繰り返す．索引キーが一致するエントリが複数存在することがあるため，このように二段階の比較になっている．

3. ハッシュ結合の実現方法

基本的な流れは自動索引の仕組みを継承する．SQLite への改造箇所は次の 4 箇所である．

(1) 新たなオペコードの追加
ハッシュ結合を実現するためのオペコードを 4 つ作成する（*HashOpen*，*HashInsert*，*HashSearch*，*HashNext*）．

(2) 自動索引生成
自動索引の代わりにハッシュ表を生成する処理に置き換える．

(3) 索引からの探索と結合演算
索引を探索するオペコードをハッシュ用に変更する．さらに，ハッシュ衝突を考慮して実データ同士の比較を行うようにする．

(4) カーソル移動
ハッシュ値が一致する行へカーソルを移動させるようにする．

3.1. 新たに追加するオペコード

以下のオペコードを追加する．

- *HashOpen*
  ハッシュ表を格納する場所を用意する．
- *HashInsert*
  ハッシュ結合のキーとなるカラム値と行番号をハッシュ表に登録する．
- *HashSearch*
  探索キーの値を入力として，ハッシュ値を計算して，ハッシュ表からハッシュ値が一致するエントリを探す．見つかったエントリからテーブルの行番号を特定し，入力テーブルのカーソルをその行に移動させる．
- *HashNext*
  HashSearch で見つかったエントリは複数存在することがあるため，次のエントリが指す行番号に入力テーブルのカーソルを移動させる．

これらのオペコードを使用して，ハッシュ結合用の実行計画を生成できるプランナへの改造方法を以下に述べる．

Copyright © 2015 by Information Processing Society of Japan and
The Institute of Electronics, Information and Communication Engineers
All rights reserved.

3.2.ハッシュ表の作成

ハッシュ表の生成には，索引の格納領域の確保とカーソルの作成を行う *OpenAutoindex* の代わりにハッシュ表の格納領域とカーソルの作成を行う *HashOpen* を，索引にエントリを登録する *MakeRecord* と *IdxInsert* の代わりにハッシュ表にエントリを登録する *HashInsert* を使用する（図 2 右上）．

さらに，エントリに含めるカラムも変更する．*HashInsert* の入力となるカラム値は，*Column* で取得するが，複数カラムを入力とする場合は図 2の※2 の *Column* が複数連続することになる．

2.2.1で述べたとおり自動索引による結合では結合キー以外のカラムもエントリに含んでいる．ハッシュ結合では結合キーと行番号からエントリを作成してハッシュ表に登録する．このように変更する理由はエントリを作成するコストを抑えるためである．結合条件に一致した行のカラム値を取得にはハッシュのビルド入力用カーソルを使用する．

HashInsert でエントリを作成しハッシュ表に登録する処理において，2.2.2で述べたとおり異なる型のデータが混在するため，エントリを作成する前にカラム値の型変換が必要である．カラム値は比較演算の Type Affinity に従って必要な場合のみ型を変換する．Type Affinity は結合条件式から判定でき，その判定機能は SQLite が持つ．

3.3.ハッシュ表からの探索

まずハッシュ表を探索し，ハッシュ値が一致した場合，さらに実データ同士を比較する（図 3）．そのために，*SeekGe* の代わりに *HashSearch* を使用してハッシュ表から探索するようにする（図 2右下）．もし結合条件の一方がカラムではなく式だった場合，*Column* ではなくその式を計算するオペコードが入ることになる．それから，*IdxGT* の代わりに *Column* と *Ne* を使用してハッシュ値が一致した行について実データでの値を比較する．

HashSearch において，探索キーは必要に応じて型変換してからハッシュ表から探索する．変換ルールは3.2と同様に Type Affinity に従う．

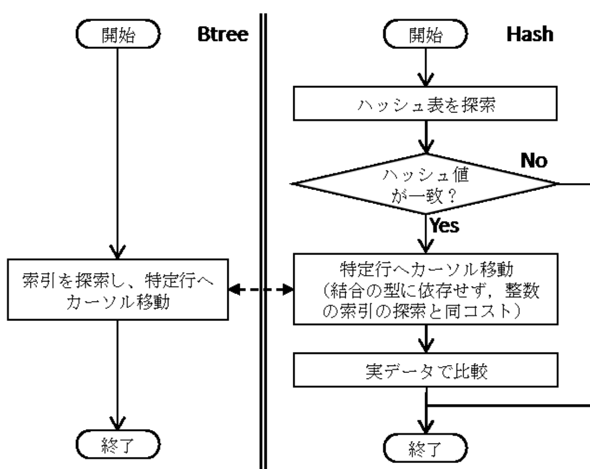

図3 探索処理の流れ

探索した結果，ハッシュ値が一致しなければその行の結合処理は終了するが，一致するエントリが見つかった場合，実データでの比較を行う．ハッシュ表探索により該当の行番号が得られ，その行にカーソルを移動させる．SQLite は行番号を整数で管理していて，指定した行番号へのカーソル移動は整数値の索引に対する探索と同等の処理になる．もし結合カラム値が文字列の場合，自動索引による結合では文字列による探索処理になるが，ハッシュの場合は常に整数で探索（行番号による探索）することになる．

3.4.カーソルの移動

SQLite の索引はキーで整列されているため 1 つずつずらすだけでよい．一方，ハッシュ結合の場合は移動方法が異なり，ハッシュ値が一致する行を抽出する必要がある．そのために，*Next* の代わりに *HashNext* を使用するようにする．

4. 実験

ハッシュ結合の効果を確認するために実験を行った．

4.1.実験内容

結合処理に要する時間を測定した．結合列は整数型と文字列型の 2 通りに対して，表 2に示すようにテーブルのサイズを変えて実験した．

4.1.1. 実験に使用したデータとクエリ

2 つのテーブル t1 と t2 を用意した．スキーマは次のとおりである．

CREATE TABLE t1 (a INTEGER, b INTEGER, c TEXT);
CREATE TABLE t2 (d INTEGER, e INTEGER, f TEXT);

a,d 列には 1 から昇順に整数を，b,e 列にはランダムな整数を（b は 5 桁，e は 6 桁），c,f 列には b,e 列の数値を英語化した文字列を格納しておく（表 1）．

表1 テーブル t1 のレコード例

a	b	c
1	13153	’thirteen thousand one hundred fifty three’
2	75560	’seventy five thousand five hundred sixty’
3	45865	’forty five thousand eight hundred sixty five’

次の 2 種類の SELECT 文をそれぞれ 10 回ずつ実行してその平均値を取得した．

SELECT count(a) FROM t1 INNER JOIN t2 ON b = e;
SELECT count(a) FROM t1 INNER JOIN t2 ON c = f;

この SQL 文ではテーブル t2 に対して自動索引やハッシュ表が作成される．

データベースはテーブルのサイズが異なる 4 種類を用意した（表 2）．

表2 データセット

Pattern	t1(件数)	t2(件数)	count(a)値
A	5000	5000	52
B	10000	10000	194
C	50000	50000	4996
D	100000	100000	19911

Copyright © 2015 by Information Processing Society of Japan and
The Institute of Electronics, Information and Communication Engineers
All rights reserved.

4.1.2. 実験の設定

ハッシュサイズは十分大きく設定した．ハッシュ結合用に用意したハッシュ管理モジュールは，メモリ上に乗り切る前提で実装してある．ファイルへのデータ退避による自動索引結合での性能低下を排除するために，SQLite のキャッシュサイズを十分大きくして実験を行った．

4.2.実験環境

- CPU:Core i7-2600 3.4GHz
- メモリ:8GB
- ストレージ:1TB HDD
- OS:CentOS 6.6
- SQLite バージョン:3.8.7.4

4.3.実験結果

4.3.1. 全体の処理時間

4.1.1で示した SELECT 文の実行時間を表 3に示す．Btree とは SQLite オリジナルの自動索引によるネステッドループ結合のことである．

表3 結合処理時間（ミリ秒）

Pattern	INTEGER		TEXT	
	Btree	Hash	Btree	Hash
A	9.50	9.31	17.2	11.2
B	17.0	13.1	31.7	14.5
C	77.1	35.4	160	43.1
D	160	74.8	354	92.8

今回使用したデータセットでは，いずれのパターンでもハッシュ結合の方が高速であった．データ量が多いほど性能差は顕著で，ハッシュ値が分散したことにより高速に探索できたと考えている．なお，いずれの場合も索引を使用せずフルサーチした場合よりも 100 倍以上高速である．

また，整数の結合より文字列の結合の方がハッシュ結合の効果が顕著であることが分かった．理由としては，ハッシュ値が一致した場合，該当の行へのカーソル移動が整数も文字列も同じコストで行えるからであると考えている（図 3）．索引の場合は整数より文字列の場合の方が高コストである．つまり，特定行へのカーソル移動のみを考えると，整数値の索引に対する探索と同等の処理なため整数の結合ではカーソル移動処理では差がつかず，文字列の結合では差がつく．全体で見るとハッシュ表の突き合わせをする分ハッシュ結合の方が余計にコストがかかる一方で，ハッシュ値の一致するエントリが見つからなければカーソル移動しないため Btree より低コストとなる．

4.3.2. 処理時間の内訳

索引やハッシュ表の作成時間と探索時間の内訳を調査した．作成時間は *OpenAutoindex* あるいは *HashOpen* から *Next* ループの最後までの時間，探索時間は作成終了時点から *ResultRow* で結果を生成し終わるまでの時間とした．

結果を表 4に示す．括弧内は全体の処理時間に対する割合である．作成も探索もハッシュの方が短時間で実行できていることが分かった．データ量が多いほど Btree へのデータ登録位置を決定するコストが大きいことが性能劣化の主な原因と考えている．また，探索においても Btree では木の深さが高くなる分ハッシュより遅くなっている．さらに，自動索引の作成では，結合カラム以外のカラムも索引のエントリに含んでいることがハッシュ表の作成より時間がかかった原因のひとつで，実験ではそのカラムは整数値だが，もし文字列カラムが射影対象になっていたら更に性能劣化する．

表4 処理時間の内訳（上）INTEGER（下）TEXT

Pattern	Create		Search	
INTEGER	Btree	Hash	Btree	Hash
A	7.75 (81)	8.20 (88)	1.70 (18)	1.06 (11)
B	12.9 (75)	10.9 (84)	4.05 (24)	2.07 (16)
C	54.6 (71)	22.6 (64)	22.4 (29)	12.8 (36)
D	113(70)	39.8 (53)	47.6 (30)	34.9 (47)

Pattern	Create		Search	
TEXT	Btree	Hash	Btree	Hash
A	13.6 (79)	9.78 (87)	3.60 (21)	1.37 (12)
B	23.8 (75)	11.6 (80)	7.87 (25)	2.80 (19)
C	111 (70)	26.0 (60)	48.6 (30)	17.1 (40)
D	243 (69)	48.0 (52)	111 (31)	44.8 (48)

5. あとがき

SQLite にハッシュ結合を導入し，今回のデータセットでは自動索引による結合よりも高速に結合できた．カラム値の重複による性能劣化は想像されるので，今後様々なパターンでの調査しプランナをより賢くすることで，自動索引とハッシュ結合を上手に使い分けて SQLite を更に高速化できると考えている．

今回は自動索引の置き換えという位置付けでハッシュ結合を導入した．クエリの実行時に一時的に索引を作成したりせず，ユーザがあらかじめ作成しておいた索引を使うこともある．しかし，自動索引の作成時間を除いた探索のみの処理時間よりハッシュ結合処理全体の方が高速となるパターンも存在し，本提案が効果的であった．

もしハッシュ結合においてあらかじめ作成しておいたハッシュ表を使って結合処理する仕組みを備えようとすると，ハッシュ表を永続化させてハッシュ値に応じて高速に読み込める仕組みと，ハッシュ表には格納していない実データ（カラム値）を高速に取得する仕組みが必要である．探索時間がハッシュ結合の方が高速であることが確認できたため，この仕組み次第では，ハッシュ結合の適用範囲が拡大できると考えている．

6. 参考文献

[1] SQLite.
https://www.sqlite.org/
[2] SQLite - Type Affinity.
https://www.sqlite.org/datatype3.html

以上

Copyright © 2015 by Information Processing Society of Japan and
The Institute of Electronics, Information and Communication Engineers
All rights reserved.