ハイブリッドDBシステム解説

1051090 福澤優

2011.11

1　ディレクトリ情報

システムのソースコードや必要なデータはいくつかのディレクトリに分かれて格納されている．ここでは各ディレクトリの情報を示す．

1.1　COMMONディレクトリ

COMMONディレクトリでは様々な演算で用いられる共通のデータ構造体やデータ型，クラスの定義や，テーブル，属性情報などのシステムカタログを扱うクラスのコードが格納されている．

1.2　PosValディレクトリ

演算中で扱うテーブルの各データはデータの値とテーブル中でのポジション（テーブル中の何番目のデータか）の2つで扱われる．PosValディレクトリではこのデータを扱うためのクラスが各データ型に分かれて定義されている．

1.3　NSMディレクトリ

テーブルデータを格納する際に1つのテーブルから行指向データと列指向データの両方を生成する．ここでは行指向データ生成のためのクラスが定義されている．

1.4　DSMディレクトリ

列指向データ生成のためのクラスが定義されている．列指向データでは圧縮する場合と圧縮しない場合の2パターンのデータを生成できる．圧縮はランレングス圧縮法によっておこなわれる．

1.5　InsertOperatorディレクトリ

データから行指向と列指向データを生成するときに呼び出すクラスが定義されている．またTPC-Hのベンチマークより行指向と列指向データを生成するスクリプトも格納されている．ここで生成されたデータはRunDataディレクトリに全て格納される．

1.6　Operationディレクトリ

列指向，行指向共通のSQLの演算子クラスが定義されている．行指向データに対してのSQLのSELECT命令は，このディレクトリにあるクラスを組み合わせて行うことが可能．

1.7　Operation_Cディレクトリ

列指向データ専用の演算子が定義されている．Operationディレクトリに定義されている演算子を含めて，列指向データまたは行指向と列指向を組み合わせたデータに対してSQLのSELECT命令を実行することが可能となる．

1.8　Nodeディレクトリ

選択演算を行う際に用いる演算木を生成する際に各ノードに用いられるクラスが定義されている．

1.9　Aggregationディレクトリ

GROUP BY演算で行う集約演算が定義されている．

1.10 RunData ディレクトリ

DB システムで扱うデータの全てが格納されているディレクトリ. システムカタログ, テーブル情報が BDB データとして格納されている.

1.11 TPC-H_DATA ディレクトリ

実験用データとして用いられる TPC-H の各テーブルデータがスケール 1 の大きさで格納されている. InsertOperator ディレクトリにある TPC-H データ格納のためのスクリプトは，このディレクトリにあるデータを呼び出す.

1.12 TestOpe*ディレクトリ

TPC-H のクエリ 8 のクエリを段階的に実行するためのプログラムが格納されている.

1.13 SQL ディレクトリ

TPC-H のクエリ 8 のクエリを段階的に実行するための SQL 文が格納されている.

1.14 Document ディレクトリ

DB システムに関するドキュメントを格納するためのディレクトリ.

2 データ型

システム内で扱うテーブルのデータ型を述べる. テーブルデータは演算を実行する際に，value(データの値) と position(データの位置) のセットで扱われるように PosVal ディレクトリ内で定義されている. value として扱えるデータ型は COMMON ディレクトリ内の”CommonDataDef.h”で列挙型データ DataType として定義されていて，以下のデータ型が扱える.':' の前に記述してある型がシステムとして扱う型の名前で，後ろにプログラムとして内部的に扱うデータ型を示している。

- INT 型：int
- LONG 型：long
- FLOAT 型：float
- DOUBLE 型：double
- FIXED_TEXT 型：char[dataSize]
- VARIABLE_TEXT 型：char[dataSize]
- DATE 型：char[10]

DATE 型は 10 文字の文字列として”YYYY-MM-DD”の形式で扱われる.

3 テーブルの挿入

システムではテーブル単位でのデータの挿入が可能である. 以下はテーブルの格納手順について述べていく. テーブルの挿入は InsertOperator ディレクトリ内の実行ファイル”main”で行い，実行時にコマンドラインより，格納したいテーブルデータを記述したファイル名を 1 つ与える. 実行ファイルに与えるテーブルデータファイルはレコードデータが 1 行ごとに記述されており，レコード内の属性データがセパレータ'|' で区切られている必要がある. レコード数の最大値としては内部的に position データと int 型で管理しているため INT_MAX までのデータは正常に扱えるはず (?). 実行フィルを実行するとテーブルに関する情報を対話形式で入力する. 以下に入力の流れを示す.

1. 挿入テーブル名の入力
2. 挿入テーブルの属性数の入力
3. 1 つの属性名の入力
   (以下の入力はここで指定した属性に対して)
4. 属性のデータ型を選択
   (セクション 2 で示したものより選択)
5. 属性が主キーかどうか
6. 列指向データ生成時に圧縮データも作成するか
7. データがソートされているかどうか
8. 属性が TEXT 型の場合，文字数の上限の入力
9. 未入力の属性が無くなるまで 3 から繰り返し

属性情報は，コマンドラインで指定したテーブルデータファイルに記述されている属性順と一致しなければならない.

テーブル情報入力後に，システムカタログの更新とテーブルの行指向データと列指向データが (圧縮データ作成を指定した場合は圧縮データも) 作成される. ここで更新・作成したデータはすべて RunData ディレクトリに格納される.

また，InsertOperator ディレクトリ内には TPC-H_DATA ディレクトリ内に格納されている TPC-H の各テーブルデータから，自動的に挿入を行うスクリプト (”Insert_*”) が用意されている.

4 システムカタログ

テーブル挿入時に入力されたデータはシステム内でシステムカタログの情報として，テーブル情報と属性情報に分けて RunData ディレクトリに格納される.

4.1 テーブル情報

テーブル情報はシステム内においては COMMON ディレクトリ内の”CommonDataDef.h”で定義されている以下の構造体で扱われる. ここでの tableID は各テーブルに挿入順に 1 からユニークに割り当てられるものである.

```
typedef struct TABLE_REC_STRUCT{
    unsigned int tableID;
    unsigned int attriNum;
    char tableName[MAX_NAME_LENGTH];
} TABLE_REC;
```

ユーザから得られたテーブル情報は BerkeleyDB を通して RunData ディレクトリ内の”BDB_TABLE_MANEGE_FILE.db”ファイルとして保存される. また，この時 BerkeleyDB にはキーとして unsigned int 型で扱われる tableID を渡し，データとして tableID, attriNum, tableName を順に記述したものを渡し格納する. さらにテキストデータとして”TABLE_MANEGE_FILE”ファイルにもテーブル情報を出力しており 1 つのテーブル情報を一行ずつ tableName, attriNum, tableID の順にセパレータ’|’ で区切り記述している.

4.2 属性情報

テーブル情報と同様に属性情報も BerkeleyDB を通して管理される. 内部的には COMMON ディレクトリ内の”CommonDataDef.h”で定義されている以下の構造体で属性情報が扱われる.

```
typedef struct ATTRIBUTE_REC_STRUCT{
    char attributeName[MAX_NAME_LENGTH];
    unsigned int tableID;
    unsigned int attributeID;
    DataType dataType;
    unsigned short dataLength;
    bool primary;
    bool isCompress;
    bool isSort;
} ATTRIBUTE_REC;
```

attributeID は各テーブル内の個々の属性に対してユニークに 1 から割り当てられる. つまりテーブルが異なれば重複する attributeID 存在することもあり得る. 属性情報は BerkeleyDB を通して RunData ディレクトリ内の”BDB_ATTRIBUTE_MANEGE_FILE.db”ファイルとして保存される. この時，格納は 1 つの属性情報ずつ行われ，キーとしては，unsined int 型の上位 16bit に tableID の値，下位 16bit に attributeID の値をセットした値を渡す. これによって扱えるテーブルの上限と，1 つのテーブル内で扱える属性の上限はともに $2^{16} = 65536$ となる (ただし 1 つのテーブル内での属性の上限は COMMON ディレクトリ内の”CommonDataDef.h”で定義されている MAX_ATTRIBUTE_NUM によって 256 に制限されている). また，データとしては構造体のメンバーの上から順に格納した情報を渡して格納していく. 属性情報も，テーブル情報と同じくテキストデータとしても”ATTRIBUTE_MANEGE_FILE ”ファイルに記述しており，BerkeleyDB のデータに渡す構造体メンバー順にセパレータ’|’ で区切り格納している.

5 データの格納方式

ここではテーブル挿入時に BerkeleyDB を用いてデータがどのように格納されるかを述べていく.

5.1 BerkeleyDB のチューニング

データの格納や読み込みは COMMON ディレクトリ内で定義してある BDBOpe クラスを通じて行われる. BDBOpe クラスを利用する際，同じディレクトリ内の”CommonDataDef.h”ファイルで定義されている変数を元にチューニングが行われる. 表 1 に BerkeleyDB の設定の際に用いられるチューニングパラメータを示す. これらのチューニングパラメータはデータ格納時のみならず SELECT クエリ実行時にも共通で利用される.

表 1: BerkeleyDB のチューニングパラメータ

BDB_PAGE_SIZE	BerkeleyDB のページサイズ
BDB_CACHE_SIZE_BYTE	BerkeleyDB のキャッシュサイズ (byte 単位)
BDB_CACHE_SIZE_GBYTE	BerkeleyDB のキャッシュサイズ (Gbyte 単位)

5.2 行指向データ

行指向データの格納方法は BerkeleyDB にレコード単位でデータを渡して格納していく. この時，キーとしてはレコードの position 情報（取り出したデータ順に 1 からシーケンシャルに position 情報として割り当てる）を渡す. データは 1 つのレコードデータの情報を渡す. この時 VARIABLE_TEXT 型以外のデータ型では，データ長が一定であるためシーケンシャルにデータを領域に格納できる. しかし VARIABLE_TEXT 型は可変長であるため’\0’ の終端記号を付加して格納している.

ここで作成されたデータは RunData ディレクトリ内に”テーブル名.row”というデータ名で格納される.

5.3 圧縮なし列指向データ

列指向データはデータをランレングス圧縮を用いて圧縮する場合のデータと，圧縮しない場合のデータの両方がある. 圧縮なしの列指向データは全ての列データに対して作成される. ここでは圧縮なしの列指向データの格納方法について述べる.

圧縮なし列指向データでは用意されたバッファにテーブルの先頭からデータをシーケンシャルに配置して，バッファが一杯になるか，格納データが無くなった際に BerkeleyDB にデータとしてバッファを渡すという手順で格納していく. バッファはヘッダ情報としてバッファに含まれている列データの数と，バッファに格納されているデータのうち一番先頭に格納されているデータの position の 2 つの情報を，バッファの先頭に格納する. その後ろにテーブルの列データを格納していく. また BerkeleyDB に渡すキーとしてはバッファに格納されたデータの最後のデータの position 情報を渡す.

バッファサイズのチューニングは COMMON ディレクトリ内の”CommonDataDef.h”ファイルで定義されている表 2 のパラメータで行うことが可能である.

表 2: 列指向データのチューニングパラメータ

DSM_BUFFER_SIZE	列指向データのバッファサイズ

ここで作成されたデータは RunData ディレクトリ内に”tableID.attributeID.DSM”の形式で名前が付けられ格納される.

5.4 圧縮あり列指向データ

圧縮あり列指向データでは圧縮なし列指向データ同様にバッファを利用してデータを格納していく. 圧縮あり列指向データにおいては，圧縮率を上げるために，まずデータをソートする必要性がある. ソートによって元のデータから position の値が変わってしまうためソート前の position 情報を保存しておく必要性があり，ソート前の position 情報をソート後のデータの順番で圧縮なし列指向データを用いて格納する. このデータは RunData ディレクトリ内に”tableID.attributeID-to-tableID.0.JoinIndex”の形式の名前で格納される.

ソートされたデータはランレングス圧縮を行いながら格納していく. ランレングス圧縮によりデータは (データ内容, 重複データのはじめのデータの position, 繰り返しの回数) というデータ形式になり，圧縮される

(position はソート後の位置情報をセットする. この後の処理も position はソート後のもの）. 圧縮なし列指向データと同じヘッダ情報をバッファに書き込み，その後ろに圧縮されたデータが順次格納されていく．ヘッダ情報のてバッファに含まれているデータの数は，圧縮後のデータ形式を 1 つとして数えていくので，実際に取り出せるデータではないことに注意 (デコードすると当然数が多くなる).

バッファに格納されたデータは圧縮なし列指向データ同様に BerkeleyDB に渡され RunData ディレクトリ内に”tableID.attributeID.DSM_C”として格納される.

また，バッファのサイズは圧縮なし列指向データの場合と同じになる.

6 SELECT クエリの実行

格納テーブルデータに対する SELECT クエリは Operation ディレクトリと Operation_C ディレクトリに定義されている演算子を用いて行われる．ここでは演算子の共通概念を記述していく.

6.1 Operatior クラス

多くの演算子は Operator クラスを親クラスとして継承している. よって Operator クラスは演算子の基本となるメソッドが定義されている．クエリを実行する際には，クエリ木を生成したのちに処理を実行していく. ここでは具体的にクエリ木を生成したあとに行われる Operator クラスで定義されている共通の処理の流れを述べていく.

まずクエリ木の各ノードの演算子の初期化を行うために，クエリ木のルートノードに初期化を行う init() メソッドを呼び出す. すると root 以下のノードは図 1 のような流れで同様に初期化が行われる.

図 1 では中心のノードに着目した処理の流れが示されている. あるノードが親ノードから init() メソッドが呼び出されると，まず自身の子ノードの init() メソッドを呼び出す. 子ノードは演算子によって 1 つか 2 つか異なる. その後子ノードの init() メソッドから戻ってくると，自身のノードの init() メソッドの処理を行い，init() メソッドの呼び出し元である親ノードに返る.

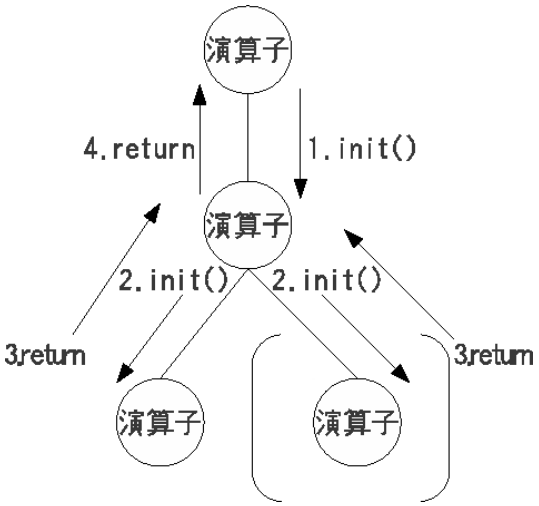

図 1: ノード初期化の流れ

次に，init() メソッドで行われる処理について述べていく.init() メソッド内で共通に行われる処理を以下に示す.

1. 子ノード init() の呼び出し
2. getAttriNum() の呼び出し
3. initAttriRec() の呼び出し
4. getAttriRec() の呼び出し
5. initPosVal() の呼び出し

まず，先ほど述べたように子ノードの init() メソッドを読み出す. 次に getAttriNum() メソッドにより子ノードから受け取るデータの属性数を得る．さらに initAttriRec() メソッドにより，先ほど得た属性数だけクラス内で保持する属性データ変数を初期化し，getAttriRec() メソッドによって子ノードから得られた属性データを格納する. 最後に initPosVal() によって属性データを参照し，子ノードから得られるタプルデータを格納する領域を初期化する.

全てのノードの初期化が行われる，つまり root ノードの init() メソッドから戻ったのちにレコードデータに対する処理を行っていく．この時，ノード共通のメソッドとして getRow() メソッドが利用される.getRow() メソッドは init() メソッドと同様の流れで，子ノードの getRow() を呼び出し処理に必要なレコードデータを子ノードから受け取り，そのレコードデータに対して

必要な処理を行い親ノードへ渡す. 子ノードは親ノードへ渡すレコードデータが無くなった場合は戻り値として'-1' を返すことになっている.

7 Operationディレクトリ内の演算子解説

ここではOperationディレクトリ内で定義されている演算子を述べていく.

7.1 Scanクラス

BerkeleyDBによって格納された行指向データから1行ずつデータを取り出し，親ノードへ値を渡す. Scanクラスは実行時の葉ノードに位置する.

7.2 OutPutクラス

子ノードから得られたデータを標準入力に全て出力する. ファイル出力や出力制限を設けるような改良が必要.

7.3 Projectionクラス

子ノードから得られたレコードデータから指定された属性データだけを取り出し親ノードへ渡す.

7.4 Selectionクラス

子ノードから得られたレコードデータから条件に一致するレコードデータのみを親ノードへ渡す. 条件の指定はNodeディレクトリ内にあるクラスを用いて演算木を生成してSelectionオブジェクトの生成時に引数として渡すことで行う.

7.5 Sortクラス

子ノードから出力される全てのデータを取り出し，指定された属性に対してソートを行う. 指定する属性は複数でもよい. 扱うレコードが少ない場合は全てメモリ上にデータを取り込んでクイックソートを行う. しかしデータ量が一定の閾値を越えるとバッファに取り込んだデータに対してソートを行い，ソートされたデータをBerkeleyDBを通して一時ファイルとして格納しておく. データが無くなるまで処理を繰り返し，親ノードにデータを渡す際に全ての一時ファイルをマージソートし，親ノードに値を渡していく. この，一時ファイルを利用する閾値 (workspace) はCOMMONディレクトリ内の"CommonDataDef.h"ファイルで定義されている (表3).

表 3: Sortクラスのチューニングパラメータ

MAX_WORK_SPACE	メモリの workspace

7.6 MargeJoinクラス

MargeJoinクラスではソートされたデータを2つの子ノードから受け取ることを想定して設計されている. 受け取ったデータ同士を指定した属性同士を比較して，一致するレコードデータ同士を結合して親ノードへ渡す.

7.7 NestedJoinクラス

2つの子ノードから受け取ったレコードデータから入れ子ループ結合を行うクラス.InnerNodeとして設定された子ノードからのレコードデータは複数のループによって何度のスキャンしなければならないため，始めにInnerNodeからのレコードデータを全てBerkeleyDBを利用して一時ファイルに格納する. その後OuterNodeとして設定された子ノードから1つずつレコードデータを受け取り，一時ファイルをスキャンしながら指定した属性データが一致した場合のみ親ノードに2つのレコードデータを結合したものを渡す.

7.8 HashJoinクラス

HashJoinクラスでは2つの子ノードから全てのレコードデータを受け取り，指定した属性同士からそれぞれのデータ集合からハッシュテーブルを作成する. ハッシュテーブルに用いるハッシュキーは指定した属性データから生成する. ハッシュテーブルはBerkeleyDBを通

してキーにハッシュ関数で作成されたハッシュキー，データとしてレコードデータを渡し作成される．

ハッシュテーブルが作成された後はハッシュキーをシーケンシャルに辿り，2 つのハッシュテーブルを参照して指定した属性同士が一致した場合のみレコード同士を結合して親ノードへ渡す．

ハッシュ関数は Operation ディレクトリの”Hash.cpp”ファイル内で定義されている．ハッシュキーは VARIABLE_TEXT 型以外のデータから生成が可能であり，ハッシュテーブルを生成する演算子では VARIABLE_TEXT 型からハッシュキーを生成するような処理はできない．

ハッシュに関するチューニングパラメータは表 4 に示す内容で，COMMON ディレクトリ内の”Common-DataDef.h”ファイルで定義されている．

表 4: ハッシュのチューニングパラメータ

HASH_SIZE	ハッシュテーブルの大きさ
MUL_NUM	ハッシュキー作成時の乗算値

7.9 OneSideHashJoin クラス

HashJoin クラスとは異なり OneSideHashJoin クラスではハッシュテーブルは 1 つしか作成しない.InnerNode として指定された子ノードに対して，指定した属性からハッシュキーを生成し，BerkeleyDB からハッシュテーブルを作成する. その後，OuterNode として指定された子ノードから 1 つずつレコードデータを取り出し，ハッシュテーブルからハッシュキーが一致するデータを参照し，指定した属性データの値が一致する場合のみ親ノードにデータを渡す．

7.10 GroupBy クラス

指定した属性データ（複数指定可能）に対してグループ化を行うクラス．グループ化は，ハッシュ法を用いて行われる. このためソートを行ってからグループ化を行う方法と比べるとデータをソートするコストがかからない. 指定した属性データよりハッシュキーを生成し，指定データの属性値のみをリストを用いたチェインハッシュ法で格納していく．ハッシュ結合の場合と異なりハッシュテーブルを BerkeleyDB を通して作成せずに，すべてメモリ上で行う実装となっているのでグループ化の対象とするデータ集合のカーディナリティが高い場合にはメモリオーバーフローが起ってしまうことに注意しなければならない．

GROUP BY 演算の際に共に行われることが多い集約演算は Aggregation ディレクトリに定義してあるクラスより演算木を生成し，GroupBy クラスに渡すことによって実行が可能である. この集約演算を行う演算木は複数指定も可能となっている. 集約演算では COUNT,MAX,MIN,SUM,AVG の演算が可能となっており，さらに Node ディレクトリ内のクラスを用いて，集約演算同士のの四則演算などが可能となっている. しかし，SQL 文で用いることができる条件を付加した集約演算（条件を満たすレコードデータのみに対して集約演算を行うなど）などの実装は出来ていない．

7.11 Extract クラス

Extract クラスは DATE 型のデータに対してのみ行われる.EXTRACT 演算によって日付データの年，月，日の中から任意のデータのみを抽出することができる. 抽出されなかったデータに対しては’0’ が割り当てられるという仕様で実装されている. 例えば”1998-05-14”から年のデータのみを抽出すると”1998-00-00”となる.Extract クラスは子ノードから受け取ったレコードデータから指定された属性に対して ECXTRACT 演算おこない，親ノードへそのレコードデータを渡す．

8 Operation_C ディレクトリ内の演算子解説

Operation_C ディレクトリには主に列指向データに対する演算子が定義されている．ここではそれらの演算子クラスについて述べていく．

8.1 ScanDSM クラス

このクラスでは非圧縮列指向データに対して行われるスキャン演算である．一行ずつデータを取り出し，ポジション情報を付加させて親ノードへデータを渡す．

8.2 ScanRLE クラス

ScanRLE クラスは圧縮列指向データに対するスキャン演算子となっている．ScanDSM クラスと異なる箇所は，圧縮データを生成する際に同時に作成した元のポジション情報データを参照して，そこからデータに対するポジション情報を付加させて親ノードへ渡す点である．

8.3 SelectBit クラス

SelectBit クラスでは Select クラス同様に条件に一致するレコードを抽出することである．しかし，SelectBit クラスでは親ノードに渡すデータとしてレコードデータを渡すのではなく，各レコードが条件に一致するかどうかを'0' と'1' で表したビットマップデータを１つずつ渡す仕様になっている．子ノードからレコードデータを受け取り，条件に照らしあせて条件に一致するレコードであれば'1' を，そうでなければ'0' をそれぞれ親ノードへと渡していく．選択条件の演算木などは Select クラスと同じように扱う．

8.4 BitSort クラス

もし圧縮データに対して SelectBit クラスの演算を実行した際にビットマップデータとして出力されるデータの順番はソート後のデータ順になっている．しかし，圧縮していないデータとのポジションの整合性を取ろうとしたときに，このビットマップの出力順をソート前のデータ順で出力する必要性が出てくる．このビットマップの並び替えを行うのが BitSort クラスである.BitSort クラスは圧縮列指向データ生成時に作成した"tableID.attributeID-to-tableID.0.JoinIndex"形式のデータを参照しながら，SelectBit クラスから渡されるビットマップをソート前の順番で親ノードへと渡す．

現在，BitSort クラスでは必ずファイルを通じてビットマップの並び替えを行っているが，データが小さい場合は全てメモリ上で並び替えを行う仕様に改良する必要がある．

8.5 BitFilter クラス

BitFilter クラスでは指定したレコードデータとビットマップを出力する２つの子ノードから，ビットマップに一致するデータを出力する演算子である．このことより，１つの子ノードとして SelectBit クラスか BitSort クラスを設定しなければならない，また他の条件としては子ノードから受け取るレコードデータとビットマップのデータのポジション情報が一致しなければならなく，さらにデータ数も同一でなければならない．

ビットマップデータとして'1' を受け取った場合に限り，一致するレコードデータを親ノードへと渡す．

8.6 JoinIndex_OneHash クラス

JoinIndex_OneHash クラスは２つのレコードデータから Join 演算を行う演算子である．アルゴリズムとしては OneSideHashJoin クラスと同様に処理されるが，JoinIndex_OneHash クラスでは出力される値としてレコードデータが出力されるが，レコードデータの各属性のポジションデータを対応みると JoinIndex データとしても扱えるという出力になっている．

現在 JoinIndex_OneHash クラスではハッシュテーブルを作成する InnerNode からの受け取るデータとしては１属性しか持たないレコードデータと仮定して設計されている（複数レコードでも動作はするが，Join 条件のレコード属性以外のポジション情報が維持されない仕様になっている）．

8.7 ScanFromJI クラス

outerNode として設定した子ノードのレコードデータのうちから指定された属性データのポジション情報と，innerNode として渡されるデータのレコードデータのポジション情報同士を比較して，一致するレコードデータ同士を結合して親ノードへ渡す．

制約としては，innerNode,outerNode の両方から受けとるデータの順はポジションに対してソートされていなければならない．つまり，ポジションに対するマージ結合を行うような実装となっている．

また，innerNode から受け取るレコードデータは１属性のみと仮定して実装されている（複数属性でもよ

いが，ポジションを参照する属性はレコードの 1 番目の属性となる）．

8.8 ScanFromJI_OneHash クラス

ScanFromJI クラスではポジションに対してマージ結合を行うため，レコードデータが注目される属性データのポジションに対してソートされていなければならないという制約が存在した. この制約が無くてもポジション情報同士を比較する結合ができる演算子が ScanFromJI_OneHash クラスである．

OneSideHashJoin クラスと同様に innerNode 側に対してのみハッシュテーブルを作成し，結合を行っていく．ただし，このときはポジション情報からハッシュキーが生成される．

これによって，ポジション情報に対してソートされていないレコードデータ同士を結合することが可能となる．