Algoritma Brute Force dan Greedy pada Query Optimizer H2 Database Engine

Danny Andrianto 13510011
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
danny.andrianto@students.itb.ac.id
13510011@std.stei.itb.ac.id

***Abstract*—Makalah ini akan membahas pengimplementasian algoritma *brute force* (*exhaustive search*) dan *greedy* pada optimisasi *query* di H2 *database engine*.**

***Index Terms*—H2, database, optimize, query, java, brute force, exhaustive search, greedy**

I. H2 Database Engine

H2 *database engine* adalah DBMS (*Database Mangement System*) *open-sorce* yang ditulis dalam bahasa Java. DBMS sendiri merupakan suatu software yang menyediakan struktur *database* dan *interface* untuk membantu mengakses informasi yang ada pada *database* tersebut. Dengan DBMS, pengguna bisa menyimpan dan mengakses informasi pada *database* dengan mudah dan efisien. Karena ditulis dalam bahasa Java, H2 *database* memiliki kelebihan-kelebihan sebagai berikut:

- Mudah untuk diintegrasikan dengan aplikasi java
- Dapat berjalan pada *platform-platform* yang berbeda
- Memiliki *unicode support*

Dari sekian banyak fungsi yang dimiliki H2, pada makalah ini penulis hanya berfokus pada satu yaitu optimisasi *query*. Optimisasi *query* adalah proses merubah suatu masukan *query* dari pengguna ke bentuk yang paling optimalnya. H2 menentukan bentuk *query* paling optimal berdasarkan *cost* yang dibutuhkan untuk mengeksekusi *query* tersebut. Optimisasi *query* menjadi sangat penting dalam menangani *query statement* yang kompleks karena dapat mengurangi *cost* yang dibutuhkan secara signifikan.

H2 ditulis oleh Thomas Mueler, pengembang dari Hypersonic SQL, dan pertama kali dipublikasikan pada 14 Desember 2005. Saat ini H2 telah sampai pada versi 1.3.170, dengan *update* terakhir pada tanggal 30 November 2012. H2 sendiri merupakan singkatan dari Hypersonic 2 walaupun memiliki kode yang sama sekali berbeda dengan Hypersonic SQL. Saat ini H2 telah dapat men-*support* semua sistem operasi mayor seperti Windows XP, Windows Vista, Windows 7, Mac OS X, dan Linux.

II. Algoritma Brute Force

Algoritma *brute force*, sesuai namanya, adalah algoritma yang menyelesaikan masalahnya tanpa strategi khusus dan lebih mengandalkan pada kekuatan komputer yang mengeksekusi algoritma tersebut. Algoritma *brute force* adalah algoritma yang paling mudah untuk diimplementasi dan bisa digunakan hampir untuk setiap permasalahan. Kekurangan dari algoritma ini tentunya adalah pengeksekusian algoritma ini yang umumnya membutuhkan penggunaan resource yang besar dan waktu yang lama.

Sebagai contoh, kita ambil algoritma *sorting*. Jika kita ingin solusi yang mudah, kita bisa menggunakan algoritma *brute force* (*selection sort*) untuk menyelasikan masalah ini. Jika banyak elemen yang ingin di-*sort* adalah n, maka algoritma ini butuh melakukan iterasi sebanyak n kali dan dengan tiap iterasi jika dirata-rata melakukan hingga n/2 kali operasi perbandingan. Dengan kompleksitas n^2, tentunya kebutuhan akan meningkat drastis siring dengan bertambanya jumlah elemen yang ingin disort. Pada sisi lain, kita lihat solusi untuk permasalahan ini dengan algoritma *divide and conquer* (*merge sort*). Stategi khusus yang digunakan pada algoritma ini dapat meminimalkan jumlah operasi perbandingan yang dibutuhkan. Hasilnya adalah rata-rata kompleksitasnya yang senilai n.log(n) yang tentunya lebih baik daripada *selection sort*.

Salah satu implementasi dari pendekatan strategi brute force adalah *exhaustive search*. *Exhaustive search* adalah strategi brute force yang ditujukan untuk menyelesaikan masalah kombinatorik. Strategi algoritma ini berisi men-*generate* semua kombinasi solusi yang mungkin kemudian memilih satu atau lebih kombinasi yang sesuai dengan keinginan kita yang menjadi solusi. *Exhaustive search* adalah algoritma *brute force* yang akan digunakan pada pembahasan *query optimizer* ini.

Walaupun bukan merupakan algoritma yang cerdas maupun efisien, algoritma *brute force* tetap perlu dipandang sebagai suatu strategi algoritma yang penting karena beberapa alasan. Pertama, seperti telah penulis sebutkan sebelumnya, tidak seperti strategi algoritma lainnya algoritma *brute force* dapat diimplementasi pada

hampir setiap permsalahan yang ada. Selain itu, terkadang mendesain suatu algoritma yang rumit mungkin tidak dibutuhkan jika permasalahan yang ada dapat diselesaikan dengan algoritma *brute force* dengan *cost* yang dapat diterima. Terakhir, algoritma *brute force* dapat juga digunakan sebagai patokan untuk menentukan efisiensi suatu algoritma.

"When in doubt, use brute force" - Ken Thompson

III. Algoritma Greedy

Algoritma *greedy* merupakan strategi algoritma yang menghasilkan solusi melalui tahap-tahap yang mana tiap tahap menghasilkan suatu solusi parsial hingga solusi lengkap diperoleh. Pada tiap tahap, algoritma melakukan hal yang sesuai namanya greedy yatu memilih pilihan yang paling baik saat itu (optimum lokal) dengan harapan solusi yang dicapai nanti merupakan solusi terbaik (optimum global).

Setelah membuat pilihannya pada satu tahap, algoritma *greedy* tidak bisa mundur ke tahapan sebelumnya. Hal ini membuat algoritma *greedy* memilki satu kelemahan yaitu solusi akhir yang dicapai – walaupun tiap pilihan optimum lokal – belum tentu optimum global. Implementasi algoritma *greedy* juga terbatas pada permasalahan optimisasi.

Suatu algoritma *greedy* harus memiliki elemen-elemen sebagai berikut:

- Himpunan kandidat
- Himpunan seleksi
- Fungsi seleksi
- Fungsi kelayakan
- Fungsi obyektif

Dari elemen-elemen yang disebutkan di atas, bisa disimpulkan bahwa algoritma greedy adalah algoritma greedy yang mencari suatu himpunan solusi yang tersiri dari elemen-elemen yang ada pada himpunan kandidat dengan menggunakan fungsi seleksi dan fungsi kelayakan untuk memilih elemen-elemennya dan dioptimisasi dengan fungsi obyektif.

Skema umum algoritma greedy:

```
function greedy(input C: himpunan_kandidat) → himpunan_kandidat
{ Mengebalikan solusi dari persoalan optimasi dengan algoritma greedy
Masukan: himpunan kandidat C
Keluaran: himpunan solusi yang bertipe himpunan_kandidat }
Deklarasi
      x: kandidat
      S: himpunan_kandidat
Algoritma
      S ← {} { Inisialisasi S dengan kosong }
      while (not SOLUSI(S) and (C != {})) do
            x ← SELEKSI(C)
            C ← C – {X}
            if LAYAK(S U {X}) then
                  S ← S U {X}
            endif
      endwhile
      {SOLUSI(s) or C = {}}
      if SOLUSI(S) then
            return S
      else
            write('tidak ada solusi')
      endif
```

Berikut adalah contoh-contoh algoritma yang menerapkan strategi *greedy*:

- Algoritma Prim: membuat spanning tree minimum dari graf berarah
- Algoritma Kruskal: membuat spanning tree minimum dari graf berarah
- Algoritma Dijkstra: menyelesaikan masalah shortest path dari satu sumber
- Pohon Huffman: pohon biner yang meminimalkan bobot jarak dari akar ke daun untuk suatu tingkat
- Kode Huffman: salah satu aplikasi dari pohon Huffman, meng-assign suatu kumpulan bit untuk merepresantasikan suatu simbol berdasarkan frekuensi kemunculannya di text

"Greed, for lack of a better word, is good! Greed is right! Greed Works!" - Michael Douglas, *Wall Street*, 1987

IV. Algoritma Optimizer pada H2

Optimisasi pada H2 dilakukan oleh sutu kelas bernama Optimizer memiliki konstanta-konstanta sebagai berikut:

```
  private static final int
MAX_BRUTE_FORCE_FILTERS = 7;
  private static final int MAX_BRUTE_FORCE =
2000;
```

MAX_BRUTE_FORCE_FILTERS mendefinisikan jumlah tabel maksimal yang dapat dibuat menggunakan *brute force*. MAX_BRUTE_FORCE mendefinisikan jumlah *plan* maksimal yang dapat dihtung *cost*-nya.

Selain konstanta-konstanta di atas, kelas Optimizer memiliki atribut sebagai berikut:

```
  private TableFilter[] filters;
  private Expression condition;
  private Session session;

  private Plan bestPlan;
  private TableFilter topFilter;
  private double cost;
```

Atribut-atribut di atas menyimpan variabel-variabel yang dibutuhkan untuk optimisasi. Filters menyimpan tabel-tabel yag terlibat, condition menyimpan ekspresi-ekspresi yang ada, session untuk menyimpan sesi koneksi user dengan database, bestPlan akan meyimpan plan hasil optimisasi, topFilter menyimpan tabel teratas yang ada pada stack filters, dan cost akan menyimpan *cost plan* terbaik.

Unutk mengoptimisasi suatu *query*, kelas Optimizer akan meng-*invoke* suatu *method* bernama optimize.

```
    /**
     * Calculate the best query plan to use.
     */
    void optimize() {
        calculateBestPlan();
        bestPlan.removeUnusableIndexConditions();
        TableFilter[] f2 = bestPlan.getFilters();
        topFilter = f2[0];
        for (int i = 0; i < f2.length - 1; i++) {
            f2[i].addJoin(f2[i + 1], false, false,
null);
        }
        for (TableFilter f : f2) {
            PlanItem item = bestPlan.getItem(f);
            f.setPlanItem(item);
        }
    }
```

Method inilah yang kemudian akan meng-*invoke method* caculateBestPlan yang akan mendapatkan akan mengisi atribut bestPlan sebelum *plan* tersebut diimplementasikan pada *query*. *Method* calculateBestPlan inilah yang akan mengimplementasikan algoritma *brute force* dan *greedy*.

```
    private void calculateBestPlan() {
        start = System.currentTimeMillis();
        cost = -1;
        if (filters.length == 1) {
            testPlan(filters);
        } else if (filters.length <=
MAX_BRUTE_FORCE_FILTERS) {
            calculateBruteForceAll();
        } else {
            calculateBruteForceSome();
            random = new Random(0);
            calculateGenetic();
        }
    }
```

Bisa kita lihat dari *source code* di atas, dalam mengptimisasi *query*, H2 menggunakan algoritma *brute force* (calculateBruteForceAll) untuk *query-query* yang sederhana dan *query* dengan kompleksitas sedang. Definisi dari kompleksitas sedang adalah query yang hanya melibatkan 7 (MAX_BRUTE_FORCE_FILTERS) atau kurang tabel pada operasi *join*. Dengan algoritma ini, estimasi *cost* dari semua plan yang mungkin akan dihitung dan dicari nilai minimalnya.

```
    private void calculateBruteForceAll() {
        TableFilter[] list = new
TableFilter[filters.length];
        Permutations<TableFilter> p =
Permutations.create(filters, list);
        for (int x = 0; !canStop(x) && p.next();
x++) {
            testPlan(list);
        }
    }
```

Algoritma di atas merupakan algoritma *exhaustive search*. Hal ini dapat dilihat dari tahapan algoritma di atas yang dimulai dengan membuat semua alternatif plan yang mungkin. Kamudian semua plan tersebut akan dibandingkan untuk mencari *plan* dengan *cost* terkecil.

Seperti telah penulis tulis sebelumnya, walaupun dapat memberikan jawaban yang tepat, algoritma *brute force* akan memberikan masalah jika *query* masukan *user* cukup kompleks sehngga menghasilkan ribuan alternatif *plan*. Tiap iterasi akan memanggil method testPlan yang akan mengisi bestPlan dengan *plan* yang *cost*-nya paling kecil.

```
    private boolean testPlan(TableFilter[] list) {
        Plan p = new Plan(list, list.length,
condition);
        double costNow = p.calculateCost(session);
        if (cost < 0 || costNow < cost) {
            cost = costNow;
            bestPlan = p;
            return true;
        }
        return false;
    }
```

Walaupun *method* di atas sekilas terlihat hanya *method* perbandingan biasa, namun jika kita lihat baik-baik untuk mendapatkan *cost* dari suatu *plan* dipanggil *method* caculateCost dari kelas Plan.

```
    /**
     * Calculate the cost of this query plan.
     *
     * @param session the session
     * @return the cost
     */
    public double calculateCost(Session session) {
        double cost = 1;
        boolean invalidPlan = false;
        int level = 1;
        for (TableFilter tableFilter : allFilters)
{
            PlanItem item =
tableFilter.getBestPlanItem(session, level++);
            planItems.put(tableFilter, item);
            cost += cost * item.cost;
            setEvaluatable(tableFilter, true);
            Expression on =
tableFilter.getJoinCondition();
            if (on != null) {
                if (!
on.isEverything(ExpressionVisitor.EVALUATABLE_VIS
ITOR)) {
                    invalidPlan = true;
                    break;
                }
            }
        }
        if (invalidPlan) {
            cost = Double.POSITIVE_INFINITY;
        }
        for (TableFilter f : allFilters) {
            setEvaluatable(f, false);
        }
        return cost;
    }
```

Method calculateCost ini tidak akan dibahas, poinnya adalah bisa dilihat dengan jelas bahwa perhitungan *cost* bukanlah sesuatu yang mudah dan dapat dilakukan berulang kali tanpa *cost* yang besar. Menghitung ribuan plan dan membandingkannya satu-satu tentu akan memakan waktu lama dan *cost* yang tidak sedikit. Untuk itulah, algoritma *greedy* digunakan sebagai alternatif jika

query cukup rumit. Kembali ke *method* caclulatebestPlan sebelumnya, terlihat bahwa jika jumlah tabel yang terlibat lebih dari 7, *method* yang dipanggil adalah calculateBruteForceSome, bukan calculateBruteForceAll.

```
  private void calculateBruteForceSome() {
      int bruteForce =
getMaxBruteForceFilters(filters.length);
      TableFilter[] list = new
TableFilter[filters.length];
      Permutations<TableFilter> p =
Permutations.create(filters, list, bruteForce);
      for (int x = 0; !canStop(x) && p.next();
x++) {
          // find out what filters are not used
yet
          for (TableFilter f : filters) {
              f.setUsed(false);
          }
          for (int i = 0; i < bruteForce; i++) {
              list[i].setUsed(true);
          }
          // fill the remaining elements with
the unused elements (greedy)
          for (int i = bruteForce; i <
filters.length; i++) {
              double costPart = -1.0;
              int bestPart = -1;
              for (int j = 0; j <
filters.length; j++) {
                  if (!filters[j].isUsed()) {
                      if (i == filters.length -
1) {
                          bestPart = j;
                          break;
                      }
                      list[i] = filters[j];
                      Plan part = new Plan(list,
i+1, condition);
                      double costNow =
part.calculateCost(session);
                      if (costPart < 0 ||
costNow < costPart) {
                          costPart = costNow;
                          bestPart = j;
                      }
                  }
              }
              filters[bestPart].setUsed(true);
              list[i] = filters[bestPart];
          }
          testPlan(list);
      }
  }
```

Pada *method* ini, penggunaan algoritma *brute force* dibatasi. Algoritma *brute force* digunakan hanya untuk menghitung sebagian dari *plan*. Dengan *method* getMaxBruteForceFilters, tabel yang digunakan untuk membuat *plan* dibatasi terlebih dahulu agar *plan* yang dihasilkan tidak melebihi 2000 (MAX_BRUTE_FORCE) buah.

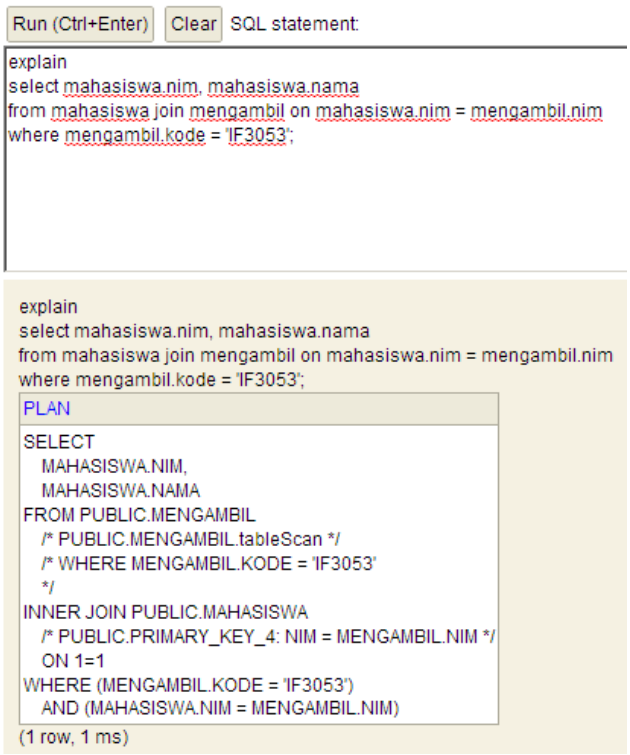
```
  /**
   * How many filter to calculate using brute
force. The remaining filters are selected using a
greedy algorithm which has a runtime of (1 + 2
+ ... +
   * n) = (n * (n-1) / 2) for n filters. The
brute force algorithm has a runtime of n * (n-1)
* ... * (n-m) when calculating m brute force of n
total. The combined runtime is (brute force) *
(greedy).
   *
   * @param filterCount the number of filters
total
   * @return the number of filters to calculate
using brute force
   */
  private static int getMaxBruteForceFilters(int
filterCount) {
      int i = 0, j = filterCount, total =
filterCount;
      while (j > 0 && total * (j * (j - 1) / 2)
< MAX_BRUTE_FORCE) {
          j--;
          total *= j;
          i++;
      }
      return i;
  }
```

Walaupun jumlah *plan* yang dihitung sudah dibatasi, *plan* yang ada masih belum lengkap karena *method* getMaxBruteForceFilters membatasi jumlah tabel yang dimasukkan ke *plan*. Untuk itulah, kemudian digunakan algoritma *greedy* untuk melengkapi *plan* yang sudah ada. Di *method* calculateBruteForceSome bisa dilihat pengisian dilakukan tiap bagian (part) dengan memilih bagian yang memiliki *cost* terkecil saat itu menggunakan iterasi biasa. Hal ini sesuai dengan algoritma *greedy* dimana tiap tahap diisi dengan memilih solusi terbaik pada saat itu saja dan tidak bisa kembali ke tahap sebelumnya.

Pada *method* calculateBruteForceSome, himpunan kandidat adalah variabel filters, himpunan solusi adalah variabel list, fungsi seleksi adalah memilih part dengan cost minimum, fungsi kelayakan adalah memerikasa apakah tabel yang dipilih sudah sesuai dengan tabel yang tersedia, dan fungsi obyektifnya adalah cost plan yang dibuat minimum.

Mungkin optimasi ini tidak menghasilkan *query* yang paling optimal, tetapi cara ini cukup efektif untuk melakukan hal ini dan besar kemungkinannya *plan* yang didapat adalah yang paling optimal. Seperti calculateBruteForceAll, di akhir calculateBruteForceSome juga dipanggil method testPlan untuk mencari plan terbaik dengan cara *brute force*.

Terakhir, penulis akan menunjukkan kerja *query optimizer* dengan melampirkan *plan* eksekusi dari suatu *query*. Penulis hanya kan melampirkan *plan eksekusi*, tidak hasilnya. Pada contoh ini *database* yang digunakan memiliki 3 tabel. Tabel pertama bernama mahasiswa yang menyimpana data seluruh mahasiswa yang ada dengan atribut NIM dan nama mahasiswa. Tabel kedua yang bernama kuliah menyimpan data kuliah yang tersedia dengan atribut kode mata kuliah dan nama dari kulish tersebut. Tabel terakhir merupkaan sebuah relasi yang berisi data pengambilan mata kuliah oleh mahasiswa. Tabel ini memiliki atribut NIM mahasiswa yang besangkutan dan kode mata kuliah yang diambilnya. *Query* yang akan dicoba adalah query untuk menampilkan mahasiswa yang mengambil kuliah dengan kode IF3053.



Gambar di atas menunjukkan *plan* yang dibuat oleh DBMS H2 setelah melalui optimisasi. Kita bisa lihat sedikit perbedaan dari *query* masukan. Perbedaan pertama ada pada operasi *join* yang telah diubah. Pada *query* masukan, penulis memasukkan "mahasiswa join mengambil" namun plan menunjukkan bahwa yang dilakukan adalah "PUBLIC.MENGAMBIL INNER JOIN PUBLIC.MAHASISWA". Pertama, kata inner muncul karena di H2 operasi *join* ada 2 macam yaotu *inner join* atau *outer join.* Operasi *inner join* dipilih karena operasi ini mengeliminasi *tuple* dari dua tabel yang tidak memiliki pasangan. Lalu, yang cukup penting adalah urutan *join* berubah menjadi "mengambil join mahasiswa". Ini disebabkan hasil optimisasi *query* H2 yang mengestimasi bahwa "mengambil join mahasiswa" akan menghabiskan *cost* yang lebih sedikit dibandingkan "mahasiswa join mengambil". Perbedaan berikutnya ada setelah operasi *join* terjadi. Pada *query* masukan, penulis mendefinisikan untuk melakukan *join* terhadap kondisi mahasiswa.nim = mengambil.nim terlebih dahulu sebelum mengambil *tuple* yang atribut kode kuliahnya IF3053. Namun, pada *plan* eksekusi, yang terjadi adalah langung menseleksi *tuple* dengan kondisi yang merupakan gabungan dari kedua kondisi yang dimasukkan yaitu "mengambil.kode = 'IF3035' and mahasiswa.nim = mengambil.nim". Tidak butuh pengetahuan tinggi untuk memahami mengapa *query optimizer* memutuskan untuk memilih *plan* ini. Alasannya tentu karena melakukan *scan* terhadap tabel dua kali untuk dua kondisi yang berbeda tentu memakan waktu yang lebih lama dibandungkan hanya melakukan *scan* sekali dengan kondisi yang merupakan gabungan dari kedua kondisi tersebut.

V. Kesimpulan

H2 Database Engine menggunakan 2 strategi algoritma dalam pengimplementasian optimisasi *query* yaitu *brute force* dan *greedy*. Kedua algoritma tersebut digunakan sesuai dengan kekurangan dan kelebihannya masing-masing.

Algoritma *brute force* yang selalu dapat memberikan jawaban yang tepat namun membutuhkan sumber daya dan waktu yang jauh lebih banyak untuk permasalahan kompleks digunakan untuk optimisasi query sederhana dan hanya memiliki sedikit plan yang mungkin.

Algoritma greedy yang dapat menyelesaikan permasalahan dengan lebih efektif namun tidak selalu dapat memberikan jawaban terbaik digunakan untuk membantu perhitungan *query* rumit yang dapat menghasilkan puluh-ribuan *plan* yang mungkin jika murni hanya menggunakan algoritma *brute force*.

Query optimizer tidak akan merubah hasil output dari *query* masukan, melainkan hanya mengubah plannya untuk menghasilkan waktu eksekusi minimal.

Referensi

[1] *http://www.h2database.com*. Waktu akses terakhir: 21 Desember 2012 pukul 10:10.
[2] *H2 Database Engine, Version 1.3.168 (2012-07-13)*.
[3] *H2 Documentation, Version 1.3.169 (2012-09-09)*.
[4] Levitin, Anany. 2011. *Introduction to the Design and Analysis of Algorithms, 3rd Edition*. Pearson Education.
[5] Silberschatz, Korth, dan Sudarshan. 2001. *Database System Concepts, 4th Edition*. The McGraw-Hill Companies.
[6] Munir, Rinaldi. 2011. *Bahan Kuliah IF3051 Strategi Algoritma: Algoritma Brute Force*.
[7] Munir, Rinaldi. 2011. *Bahan Kuliah IF3051 Strategi Algoritma: Algoritma Greedy*.

Pernyataan

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 21 Desember 2012

Danny Andrianto 13510011