Μελέτη απόδοσης ερωτήσεων – φυσικός σχεδιασμός

LAB30237663

A)

110 φοιτητές

Χωρίς την χρήση ευρετηρίων.

	QUERY PLAN text
1	Seq Scan on "Student" (cost=0.00..4.65 rows=8 width=165) (actual time=0.028..0.114 rows=9 loops=1)
2	Filter: ((surname >= 'MA'::bpchar) AND (surname <= 'MO'::bpchar))
3	Rows Removed by Filter: 101
4	Planning time: 0.173 ms
5	Execution time: 0.133 ms

Ο optimizer επιλέγει sequential scan και φιλτράρει όλες τις σειρές βάση του φίλτρου. Το planning time σταθεροποιείται γύρο στα 0.170 ms και το execution time στα 0.130 ms.

Για το indexing επιλέγουμε btree πάνω στο επίθετο επειδή το sql ερώτημα ψάχνει εύρος πάνω σε αυτό. Το hash δεν μπορεί να εξυπηρετήσει ερωτήματα τέτοιου τύπου.

Με btree index χωρίς cluster.

	QUERY PLAN text
1	Bitmap Heap Scan on "Student" (cost=4.22..7.34 rows=8 width=165) (actual time=0.038..0.043 rows=9 loops=1)
2	Recheck Cond: ((surname >= 'MA'::bpchar) AND (surname <= 'MO'::bpchar))
3	Heap Blocks: exact=3
4	-> Bitmap Index Scan on student_surname_idx (cost=0.00..4.22 rows=8 width=0) (actual time=0.030..0.030 rows=9 loops=1)
5	Index Cond: ((surname >= 'MA'::bpchar) AND (surname <= 'MO'::bpchar))
6	Planning time: 0.218 ms
7	Execution time: 0.079 ms

Με την χρήση του btree μειώνεται το execution time σε περίπου 0.080 ms αλλά αυξάνεται το planing time στα 0.220 ms. Επειδή ο πίνακας είναι πολύ μικρός, το index επιταχύνει το execution ίσα με την αύξηση του planing time. Άρα δεν υπάρχει κέρδος.

Με btree index και χρήση cluster βάση αυτού. Το clustering έγινε πάνω στο επίθετο, επειδή το ερώτημα αναφέρεται σε αυτό.

	QUERY PLAN text
1	Bitmap Heap Scan on "Student" (cost=4.22..7.34 rows=8 width=165) (actual time=0.032..0.033 rows=9 loops=1)
2	Recheck Cond: ((surname >= 'MA'::bpchar) AND (surname <= 'MO'::bpchar))
3	Heap Blocks: exact=1
4	-> Bitmap Index Scan on student_surname_idx (cost=0.00..4.22 rows=8 width=0) (actual time=0.026..0.026 rows=9 loops=1)
5	Index Cond: ((surname >= 'MA'::bpchar) AND (surname <= 'MO'::bpchar))
6	Planning time: 0.199 ms
7	Execution time: 0.068 ms

Το cluster επιταχύνει την εκτέλεση κατά 0.010 ms και το index έχει αρχίσει να γίνεται ελάχιστα πιο αποδοτικό από τη σειριακή αναζήτηση. Επίσης στο actual time η καλύτερη και η χειρότερη περίπτωση είναι σχεδόν ίδιες. Η βελτίωση γίνεται γιατί οι πλειάδες είναι ταξινομημένες βάση του επιθέτου και βρίσκοντας το πρώτο, μπορεί να συνεχίσει σειριακά για να βρει τα υπόλοιπα. Ακόμα τα block μνήμης που διασχίστηκαν, μειώθηκαν από 3 σε 1.

Hash index.

	QUERY PLAN text
1	Seq Scan on "Student" (cost=0.00..4.65 rows=8 width=165) (actual time=0.029..0.112 rows=9 loops=1)
2	Filter: ((surname >= 'MA'::bpchar) AND (surname <= 'MO'::bpchar))
3	Rows Removed by Filter: 101
4	Planning time: 3.210 ms
5	Execution time: 0.151 ms

Το hash index δεν μπορεί να χρησιμοποιηθεί για ερωτήματα τέτοιου τύπου και ο optimizer κάνει sequential scan.

Λόγο του μικρού μεγέθους του πίνακα, κανένα ευρετήριο δεν είναι αρκετά αποδοτικό.

Διαφορές μεταξύ ευρετηρίων:

- Η πολυπλοκότητα εύρεσης μια πλειάδας του hash table είναι O(1) ενώ του btree O(logn).
- Το hash table δεν μπορεί να απαντήσει σε ερωτήματα εύρους αντίθετα με τα btree.
- Κάνοντας cluster τον πίνακα βελτιώνεται η απόδοση του btree ενώ η απόδοση του hash table δεν επιρεάζεται.
- Τα btree συντηρούνται πιο εύκολα σε αλλαγές στους πίνακες και database crash. Αντίθετα τα hash index πρέπει να αναδημιουργούνται με το REINDEX κάθε φορά. Για αυτό το λόγο η χρήση τους αποθαρρύνεται επί του παρόντος(Σύμφωνα με το manual).

500110 φοιτητές

Χωρίς την χρήση ευρετηρίων.

	QUERY PLAN text
1	Gather (cost=1000.00..19126.89 rows=27452 width=166) (actual time=1.399..169.670 rows=30938 loops=1)
2	Workers Planned: 2
3	Workers Launched: 2
4	-> Parallel Seq Scan on "Student" (cost=0.00..15381.69 rows=11438 width=166) (actual time=0.061..147.894 rows=10313 loops=3)
5	Filter: ((surname >= 'MA'::bpchar) AND (surname <= 'MO'::bpchar))
6	Rows Removed by Filter: 156391
7	Planning time: 0.311 ms
8	Execution time: 181.741 ms

Ο optimizer εκτελεί παράλληλη σειριακή αναζήτηση με 2 workers. Το μέγεθος του πίνακα δεν επηρεάζει ιδιαίτερα το planning time, αλλά γιγαντώνει τον χρόνο εκτέλεσης.

Με btree index χωρίς cluster.

	QUERY PLAN text
1	Bitmap Heap Scan on "Student" (cost=1121.81..13789.59 rows=27452 width=166) (actual time=30.212..45.779 rows=30938 loops=1)
2	Recheck Cond: ((surname >= 'MA'::bpchar) AND (surname <= 'MO'::bpchar))
3	Heap Blocks: exact=11347
4	-> Bitmap Index Scan on student_surname_idx (cost=0.00..1114.94 rows=27452 width=0) (actual time=28.244..28.244 rows=3093...
5	Index Cond: ((surname >= 'MA'::bpchar) AND (surname <= 'MO'::bpchar))
6	Planning time: 0.335 ms
7	Execution time: 46.923 ms

Το btree index βελτίωσε την αναζήτηση περίπου 4 φορές. Λόγο του μεγέθους του πίνακα το ευρετήριο είναι αποδοτικό.

Με btree index και χρήση cluster βάση αυτού.

	QUERY PLAN text
1	Bitmap Heap Scan on "Student" (cost=1121.81..13833.59 rows=27452 width=166) (actual time=26.067..30.565 rows=30938 loops=1)
2	Recheck Cond: ((surname >= 'MA'::bpchar) AND (surname <= 'MO'::bpchar))
3	Heap Blocks: exact=762
4	-> Bitmap Index Scan on student_surname_idx (cost=0.00..1114.94 rows=27452 width=0) (actual time=25.967..25.967 rows=30938 loops=1)
5	Index Cond: ((surname >= 'MA'::bpchar) AND (surname <= 'MO'::bpchar))
6	Planning time: 0.320 ms
7	Execution time: 31.210 ms

Ταξινομώντας τις πλειάδες βάση των επιθέτων, μειώθηκαν οι προσβάσεις σε blocks από 11347 σε 762. Ως αποτέλεσμα μειώθηκε ο χρόνος εκτέλεσης στα 2/3. Η χειρότερη περίπτωση του actual time πάλι πλησιάζει την καλύτερη.

Παρατηρήσεις:

- Το μέγεθος του πίνακα καθορίζει αν χρειάζεται ευρετήριο ή όχι. Αν είναι μικρό, η βελτίωση που θα προσφέρει στον χρόνο εκτέλεσης είναι ασήμαντη και ίσως μικρότερη από το overhead που προστίθεται. Αντίθετα σε πολύ μεγάλους πίνακες η αύξηση στη ταχύτητα είναι δραματική και το overhead ασήμαντο.
- Το είδος του ερωτήματος είναι ο πιο σημαντικός παράγοντας για την επιλογή ευρετηρίου. Σε ερωτήματα απλής ισότητας (=) το hash index είναι πιο αποδοτικό, ενώ σε queries εύρους (<, <=, >=, >) η καλύτερη λύση είναι το btree.
- Τέλος, το clustering ταξινομεί τα στοιχεία του πίνακα στο δίσκο βάση συγκεκριμένων attributes. Αυτά επιλέγονται ανάλογα τα ερωτήματα που γίνονται πιο συχνά ή χρειάζονται την μεγαλύτερη βελτίωση. Υποστηρίζεται μόνο από τα btree.

Β)query

```
explain analyze
select foiththsl.amka, foiththsl.amka, foiththsl.course_code, foiththsl.final_grade
from "Register" as foiththsl join "Register" as foiththsl
on foiththsl.amka < foiththsl.amka and foiththsl.register_status = 'pass'
    and (foiththsl.course_code, foiththsl.final_grade, foiththsl.register_status)
        = (foiththsl.course_code, foiththsl.final_grade, foiththsl.register_status)
```

Τα queries που γίνονται join είναι οι register με register_status = 'pass'. Πρέπει να έχουν τον ίδιο κωδικό μαθήματος και τελικό βαθμό. Τέλος ο έλεγχος 'foiththsl.amka < foitiths2.amka' υπάρχει για να μην γίνουν φοιτητές ζευγάρια με τον εαυτό τους και για να μην υπάρχουν διπλότυπα ζευγάρια.

Χωρίς ευρετήρια

merge join

	QUERY PLAN text
1	Merge Join (cost=2268.49..2344.04 rows=157 width=24) (actual time=51.453..72.045 rows=13823 loops=1)
2	Merge Cond: ((foiththsl.course_code = foithths2.course_code) AND (foiththsl.final_grade = foithths2.final_grade))
3	Join Filter: (foiththsl.amka < foithths2.amka)
4	Rows Removed by Join Filter: 18847
5	-> Sort (cost=1134.25..1145.86 rows=4644 width=24) (actual time=25.254..25.633 rows=5024 loops=1)
6	Sort Key: foiththsl.course_code, foiththsl.final_grade
7	Sort Method: quicksort Memory: 585kB
8	-> Seq Scan on "Register" foiththsl (cost=0.00..851.40 rows=4644 width=24) (actual time=0.018..7.152 rows=5024 loops=1)
9	Filter: (register_status = 'pass'::register_status_type)
10	Rows Removed by Filter: 39008
11	-> Sort (cost=1134.25..1145.86 rows=4644 width=24) (actual time=26.187..28.305 rows=32660 loops=1)
12	Sort Key: foithths2.course_code, foithths2.final_grade
13	Sort Method: quicksort Memory: 585kB
14	-> Seq Scan on "Register" foithths2 (cost=0.00..851.40 rows=4644 width=24) (actual time=0.022..7.075 rows=5024 loops=1)
15	Filter: (register_status = 'pass'::register_status_type)
16	Rows Removed by Filter: 39008
17	Planning time: 0.910 ms
18	Execution time: 73.011 ms

Ο optimizer επιλέγει πρώτα να φιλτράρει με seq scan βάση του register_status και τους δύο πίνακες. Μετά, τους σορτάρει χρησιμοποιώντας το course_code και το final_grade, ώστε να μπορεί να γίνει το merge join βάση αυτών. Ακόμα, αφαιρούνται τα διπλότυπα ζευγαριών χρησιμοποιώντας το φίλτρο "*foiththsl.amka < foithths2.amka*". Ο συνολικός χρόνος σταθεροποιείται περίπου στα 75ms. Τέλος η μνήμη που χρησιμοποιείται για το quick sort είναι περίπου 1,2MB.

Hash join

	QUERY PLAN text
1	Hash Join (cost=921.06..8454.10 rows=157 width=24) (actual time=8.517..30.727 rows=13823 loops=1)
2	Hash Cond: ((foiththsl.course_code = foithths2.course_code) AND (foiththsl.final_grade = foithths2.final_grade))
3	Join Filter: (foiththsl.amka < foithths2.amka)
4	Rows Removed by Join Filter: 18847
5	-> Seq Scan on "Register" foiththsl (cost=0.00..851.40 rows=4644 width=24) (actual time=0.022..6.637 rows=5024 loops=1)
6	Filter: (register_status = 'pass'::register_status_type)
7	Rows Removed by Filter: 39008
8	-> Hash (cost=851.40..851.40 rows=4644 width=24) (actual time=8.474..8.474 rows=5024 loops=1)
9	Buckets: 8192 Batches: 1 Memory Usage: 349kB
10	-> Seq Scan on "Register" foithths2 (cost=0.00..851.40 rows=4644 width=24) (actual time=0.009..6.845 rows=5024 loops=1)
11	Filter: (register_status = 'pass'::register_status_type)
12	Rows Removed by Filter: 39008
13	Planning time: 0.270 ms
14	Execution time: 31.273 ms

Ο optimizer επιλέγει πρώτα να φιλτράρει με seq scan βάση του register_status και τους δύο πίνακες. Μετά, φτιάχνει hash table πάνω στον δεύτερο ώστε να μπορεί να γίνει το hash join. Το join γίνεται ελέγχοντας κάθε στοιχείο του πρώτου πίνακα με τα hash που έχουν δημιουργηθεί. Ακόμα αφαιρούνται τα διπλότυπα ζευγαριών χρησιμοποιώντας το φίλτρο "*foiththsl.amka* < *foiththsl2.amka*". Ο συνολικός χρόνος σταθεροποιείται περίπου στα 30ms. Τέλος η μνήμη που χρησιμοποιείται για το hash table είναι περίπου 350kB.

nested loop join

	QUERY PLAN text
1	Nested Loop (cost=0.41..31616.10 rows=157 width=24) (actual time=0.067..493.430 rows=13823 loops=1)
2	-> Seq Scan on "Register" foiththsl (cost=0.00..851.40 rows=4644 width=24) (actual time=0.018..7.493 rows=5024 loops=1)
3	Filter: (register_status = 'pass'::register_status_type)
4	Rows Removed by Filter: 39008
5	-> Index Scan using "Register_pkey" on "Register" foiththsl2 (cost=0.41..6.61 rows=1 width=24) (actual time=0.069..0.095 rows=3 loops=5024)
6	Index Cond: ((course_code = foiththsl.course_code) AND (foiththsl.amka < amka))
7	Filter: ((register_status = 'pass'::register_status_type) AND (foiththsl.final_grade = final_grade))
8	Rows Removed by Filter: 178
9	Planning time: 0.357 ms
10	Execution time: 494.027 ms

Απενεργοποιώντας το hash join και το merge join ο optimizer υποχρεώνεται να κάνει nested loop join. Κάνοντας seq scan στον πρώτο πίνακα φιλτράρει τις πλειάδες που δεν έχουν register_status = 'pass'. Στον δεύτερο πίνακα κάνει index scan χρησιμοποιώντας το primary key και όλα τα φίλτρα. Συγκρίνει όλες τις πλειάδες του κάθε πίνακα με τις πλειάδες του άλλου. Ο συνολικός χρόνος σταθεροποιείται περίπου στα 495ms.

Παρατηρήσεις:

- Σε όλα τα join ο optimizer έχει αλλάξει τις πράξεις που κάνει. Για παράδειγμα ελέγχει ότι και τα δύο register_status είναι 'pass' , και όχι μόνο το ένα και μετά να δει ότι είναι ίδιο με το άλλο, όπως είναι γραμμένο το query.
- Ο χρόνος πλάνου είναι συγκρίσιμος σε όλους τους αλγόριθμους (0-1ms) και πολύ μικρότερος από τον χρόνο εκτέλεσης.
- Αντίθετα ο χρόνος εκτέλεσης διαφέρει σημαντικά μεταξύ των join. Το πιο γρήγορο είναι το hash join με 30ms, μετά το merge join με 75ms και τέλος το nested loop join με 495 ms. Ο optimizer διαλέγει το πρώτο, το οποίο δεν είναι το πιο αποδοτικό.
- Πέρα από τους buffer που χρειάζονται για τους ενδιάμεσους πίνακες, το πρώτο join θέλει 1.2 MB για το quicksort και το δεύτερο χρειάζεται 350kB για το hash table.
- Εξαιτίας του μεγάλου μεγέθους του πίνακα, ο χρόνος σχεδιασμού είναι πολύ μικρότερος από τον χρόνο εκτέλεσης (100 – 1000 φορές) και δεν αποτελεί σημαντικός παράγοντας για την επιλογή του κατάλληλου join.

Με ευρετήρια

merge join

	QUERY PLAN text
1	Merge Join (cost=1667.77..1743.32 rows=157 width=24) (actual time=35.754..53.005 rows=13823 loops=1)
2	Merge Cond: ((foiththsl.course_code = foiththsz.course_code) AND (foiththsl.final_grade = foiththsz.final_grade))
3	Join Filter: (foiththsl.amka < foiththsz.amka)
4	Rows Removed by Join Filter: 18847
5	-> Sort (cost=833.89..845.50 rows=4644 width=24) (actual time=17.991..18.189 rows=5024 loops=1)
6	Sort Key: foiththsl.course_code, foiththsl.final_grade
7	Sort Method: quicksort Memory: 585kB
8	-> Bitmap Heap Scan on "Register" foiththsl (cost=191.99..551.04 rows=4644 width=24) (actual time=0.450..2.244 rows=5024 loops=1)
9	Recheck Cond: (register_status = 'pass'::register_status_type)
10	Heap Blocks: exact=272
11	-> Bitmap Index Scan on register_status_idx (cost=0.00..190.83 rows=4644 width=0) (actual time=0.406..0.406 rows=5024 loops=1)
12	Index Cond: (register_status = 'pass'::register_status_type)
13	-> Sort (cost=833.89..845.50 rows=4644 width=24) (actual time=17.753..19.231 rows=32660 loops=1)
14	Sort Key: foiththsz.course_code, foiththsz.final_grade
15	Sort Method: quicksort Memory: 585kB
16	-> Bitmap Heap Scan on "Register" foiththsz (cost=191.99..551.04 rows=4644 width=24) (actual time=0.419..1.930 rows=5024 loops=1)
17	Recheck Cond: (register_status = 'pass'::register_status_type)
18	Heap Blocks: exact=272
19	-> Bitmap Index Scan on register_status_idx (cost=0.00..190.83 rows=4644 width=0) (actual time=0.380..0.380 rows=5024 loops=1)
20	Index Cond: (register_status = 'pass'::register_status_type)
21	Planning time: 0.822 ms
22	Execution time: 53.873 ms

Για το merge join χρησιμοποιήθηκαν τρία ευρετήρια. Και τα τρία είναι hash tables, πάνω στο course_code, το final_grade και το register_status. Το πιο σημαντικό είναι το table πάνω στο register_status, γιατί χρησιμοποιείται δύο φορές πάνω στον αρχικό πίνακα. Δεν γίνονται πλέον καθόλου sequential scans, αλλά μόνο heap scan. Ο νέος χρόνος εκτέλεσης είναι ~55ms και το speedup είναι περίπου 30%. Ο χρόνος σχεδιασμού παραμένει σχετικά ίδιος.

hash join

	QUERY PLAN text
1	Hash Join (cost=812.69..7853.38 rows=157 width=24) (actual time=4.669..21.273 rows=13823 loops=1)
2	Hash Cond: ((foiththsl.course_code = foiththsz.course_code) AND (foiththsl.final_grade = foiththsz.final_grade))
3	Join Filter: (foiththsl.amka < foiththsz.amka)
4	Rows Removed by Join Filter: 18847
5	-> Bitmap Heap Scan on "Register" foiththsl (cost=191.99..551.04 rows=4644 width=24) (actual time=0.459..2.000 rows=5024 loops=1)
6	Recheck Cond: (register_status = 'pass'::register_status_type)
7	Heap Blocks: exact=272
8	-> Bitmap Index Scan on register_status_idx (cost=0.00..190.83 rows=4644 width=0) (actual time=0.414..0.414 rows=5024 loops=1)
9	Index Cond: (register_status = 'pass'::register_status_type)
10	-> Hash (cost=551.04..551.04 rows=4644 width=24) (actual time=4.175..4.175 rows=5024 loops=1)
11	Buckets: 8192 Batches: 1 Memory Usage: 349kB
12	-> Bitmap Heap Scan on "Register" foiththsz (cost=191.99..551.04 rows=4644 width=24) (actual time=0.425..2.372 rows=5024 loops=1)
13	Recheck Cond: (register_status = 'pass'::register_status_type)
14	Heap Blocks: exact=272
15	-> Bitmap Index Scan on register_status_idx (cost=0.00..190.83 rows=4644 width=0) (actual time=0.387..0.387 rows=5024 loops=1)
16	Index Cond: (register_status = 'pass'::register_status_type)
17	Planning time: 0.349 ms
18	Execution time: 21.876 ms

Για το hash join χρησιμοποιήθηκαν τα ίδια ευρετήρια. Πάλι το πιο σημαντικό είναι του register_status γιατί χρησιμοποιείται δύο φορές. Ακόμα χρησιμοποιείται πάνω στον αρχικό πίνακα, ο οποίος είναι ο μεγαλύτερος. Επίσης, δεν γίνονται πλέον καθόλου sequential scans, αλλά μόνο heap scan. Ο νέος χρόνος εκτέλεσης είναι ~22ms και το speedup είναι περίπου 30%. Ο χρόνος σχεδιασμού αυξήθηκε κατά το 1/3 της προηγούμενης τιμής του, αλλά παραμένει πολύ μικρότερος από τον χρόνος εκτέλεσης.

nested loop join

	QUERY PLAN text
1	Nested Loop (cost=192.41..31315.74 rows=157 width=24) (actual time=0.504..437.454 rows=13823 loops=1)
2	-> Bitmap Heap Scan on "Register" foiththsl (cost=191.99..551.04 rows=4644 width=24) (actual time=0.455..2.228 rows=5024 loops=1)
3	Recheck Cond: (register_status = 'pass'::register_status_type)
4	Heap Blocks: exact=272
5	-> Bitmap Index Scan on register_status_idx (cost=0.00..190.83 rows=4644 width=0) (actual time=0.404..0.404 rows=5024 loops=1)
6	Index Cond: (register_status = 'pass'::register_status_type)
7	-> Index Scan using "Register_pkey" on "Register" foithths2 (cost=0.41..6.61 rows=1 width=24) (actual time=0.061..0.085 rows=3 loops=5024)
8	Index Cond: ((course_code = foiththsl.course_code) AND (foiththsl.amka < amka))
9	Filter: ((register_status = 'pass'::register_status_type) AND (foiththsl.final_grade = final_grade))
10	Rows Removed by Filter: 178
11	Planning time: 0.718 ms
12	Execution time: 438.064 ms

Σε αυτήν τη περίπτωση χρησιμοποιήθηκαν δύο ευρετήρια. Btree στο primary key και hash table στο register_status. Τα υπόλοιπα ευρετήρια αποδείχτηκαν αναποτελεσματικά καθώς διπλασίαζαν τον χρόνο εκτέλεσης. Αυτή τη φορά το hash table στο register_status χρησιμοποιείται μόνο μια φορά. Όλοι οι υπόλοιποι έλεγχοι γίνονται πάνω στα αποτελέσματα όπου επιστρέφει το btree του primary key. Ο νέος χρόνος εκτέλεσης είναι ~440ms και το speedup είναι περίπου 20%. Ο χρόνος σχεδιασμού διπλασιάστηκε, αλλά παραμένει αρκετές τάξης μεγέθους μικρότερος από τον χρόνος εκτέλεσης, με αποτέλεσμα να είναι ασήμαντος.

Παρατηρήσεις:

- Ανάλογα τον τύπο του ερωτήματος και την μέθοδο εκτέλεσης του, διαφορετικά ευρετήρια βελτιώνουν την απόδοση. Στο συγκεκριμένο, δοκιμάστηκαν διάφοροι συνδυασμοί από btrees και hash tables πάνω στις τέσσερις μεταβλητές που ελέγχονται και επιλέχτηκαν οι πιο αποδοτικοί.
- Τα πιο αποδοτικά ευρετήρια ήταν αυτά που χρησιμοποιήθηκαν πρώτα. Ο αρχικός πίνακας είναι ο μεγαλύτερος δυνατός, με αποτέλεσμα οι έλεγχοι πάνω του να είναι οι πιο χρονοβόροι.
- Τα ευρετήρια προσφέρουν αξιόλογη βελτίωση, αλλά όχι αρκετή ώστε να γίνει κάποιος άλλος τρόπος join ο πιο αποδοτικός. Το hash join παραμένει το πιο γρήγορο με ή χωρίς την χρήση τους.
- Η επιλογή του τύπου των ευρετηρίων έγινε βάση του condition που ελέγχει την κάθε μεταβλητή. Στο amka ζητείται εύρος και αποδείχτηκε πιο αποτελεσματικό το btree, ενώ στα υπόλοιπα ζητείται ισότητα και τα hash table πρόσφεραν μεγαλύτερη βελτίωση.