Παραδοτέο 2

Μάθημα: Ανάπτυξη Λογισμικού για Πληροφοριακά Συστήματα

Μέλη Ομάδας:

Ραμαντάν Κονόμι - ΑΜ: 1115201800281

Θεμιστοκλής Παπαθεοφάνους - ΑΜ: 1115202100227

Μάριος Γιαννόπουλος - ΑΜ: 1115202000032

Πίνακας Περιεχομένων

Εισαγωγή

Στα πλαίσια του δεύτερου παραδοτέου, προχωρήσαμε στη βελτιστοποίηση της διαχείρισης μνήμης και της εκτέλεσης των joins, μεταβαίνοντας από row-store σε column-store λογική για τα ενδιάμεσα αποτελέσματα. Οι βασικοί άξονες της υλοποίησης είναι η τεχνική Late Materialization για την αποδοτική διαχείριση strings, η χρήση βελτιστοποιημένων δομών ανάγνωσης (`ColumnarReader`) και η σελιδοποιημένη (paged) αποθήκευση ενδιάμεσων στηλών.

Καλύτερος Χειρισμός Strings (Late Materialization)

Σύμφωνα με το πρώτο ζητούμενο της εργασίας, έπρεπε να αλλάξουμε την αναπαράσταση των strings ώστε να αποφύγουμε τις περιττές αντιγραφές δεδομένων κατά τη διάρκεια των joins. Αντί να μεταφέρουμε τα πραγματικά δεδομένα των strings (τα οποία είναι τύπου `VARCHAR`), μεταφέρουμε μόνο μεταδεδομένα που δείχνουν στην αρχική θέση μνήμης του string.

Δομή `value_t` και Κωδικοποίηση

Όπως ορίζεται στο αρχείο `include/intermediate.h`, δημιουργήσαμε τη δομή `value_t` μεγέθους 4 bytes (32-bit), η οποία χειρίζεται ενοποιημένα τόσο τους ακεραίους (`INT32`) όσο και τα strings (`VARCHAR`).

Η υλοποίηση βασίζεται στην εξής λογική:
- Για **INT32**: Η τιμή αποθηκεύεται απευθείας στο πεδίο `value`.
- Για **VARCHAR**: Εφαρμόζουμε bit-packing στο πεδίο `value` για να αποθηκεύσουμε ένα ζεύγος `(page_idx, offset_idx)`.

Συγκεκριμένα, η συνάρτηση `encode_string` στο `intermediate.h` υλοποιεί την εξής κωδικοποίηση:
- **19 bits** για το `page_idx` (δείκτης σελίδας στον αρχικό πίνακα).
- **13 bits** για το `offset_idx` (δείκτης μέσα στη σελίδα).

```
static inline value_t encode_string(int32_t page_idx, int32_t offset_idx) {
    return {(offset_idx << 19) | (page_idx & 0x7FFFF)};
}
```

Χειρισμός NULL και Long Strings

- **NULL Τιμές**: Χρησιμοποιείται η ειδική τιμή `INT32_MIN` ως sentinel (`value_t::NULL_VALUE`).
- **Long Strings**: Εάν ένα string εκτείνεται σε πολλαπλές σελίδες, χρησιμοποιείται η ειδική τιμή `LONG_STRING_OFFSET` (`0x1FFF`) στο offset, ώστε να ενεργοποιηθεί η ειδική διαχείριση κατά το materialization.

Επιτάχυνση Προσπέλασης (ColumnarReader)

Η μετάβαση σε Late Materialization δημιουργεί την ανάγκη για γρήγορη τυχαία προσπέλαση στις αρχικές σελίδες δεδομένων. Για το σκοπό αυτό, υλοποιήσαμε την κλάση ColumnarReader (`include/columnar_reader.h`), η οποία επιλύει δύο βασικά προβλήματα:

1. **Γρήγορη Εύρεση Σελίδας (PageIndex):** Κατασκευάζουμε προ-υπολογισμένα ευρετήρια (`cumulative_rows`) που επιτρέπουν την εύρεση της σωστής σελίδας για ένα δοσμένο Row ID με χρήση Binary Search ($O(\log P)$).
2. **Πλοήγηση σε Sparse Pages:** Για σελίδες που περιέχουν NULLs (sparse), η εύρεση του n-οστού στοιχείου απαιτεί καταμέτρηση των valid bits στο bitmap. Για να αποφύγουμε τη γραμμική σάρωση, διατηρούμε `page_prefix_sums` ανά 64 εγγραφές. Έτσι, ο υπολογισμός του offset γίνεται σε $O(1)$ με χρήση της εντολής `__builtin_popcountll`.
3. **Caching:** Επειδή οι προσπελάσεις κατά το Materialization παρουσιάζουν συχνά τοπικότητα (locality), ο `ColumnarReader` αποθηκεύει (cache) την τελευταία προσπελαθείσα σελίδα. Αυτό μειώνει δραματικά το κόστος σε ακολουθιακές αναγνώσεις, καθώς αποφεύγεται πλήρως η αναζήτηση στο ευρετήριο.

Υλοποίηση Late Materialization

Η ανάκτηση των πραγματικών δεδομένων γίνεται μόνο στο τέλος της εκτέλεσης (στο `include/materialize.h`). Οι συναρτήσεις `materialize_varchar_column` χρησιμοποιούν τον `ColumnarReader` για να αποκωδικοποιήσουν τα `page_idx` και `offset_idx` και να αντιγράψουν τα strings στο τελικό buffer, ελαχιστοποιώντας το memory bandwidth κατά τα ενδιάμεσα στάδια.

Ζεύξεις σε Column Store

Το δεύτερο ζητούμενο αφορούσε τη δημιουργία μιας δομής για τα ενδιάμεσα αποτελέσματα των joins, η οποία να εκμεταλλεύεται το cache locality και να είναι συμβατή με την column-store αρχιτεκτονική.

Η Δομή `column_t`

Στο αρχείο `include/intermediate.h` ορίσαμε τη δομή `column_t`, η οποία αντικαθιστά τα row-store vectors του προηγούμενου παραδοτέου.

Τα βασικά χαρακτηριστικά της `column_t` είναι:

- **Σελιδοποίηση (Paging)**: Τα δεδομένα αποθηκεύονται σε σελίδες (`struct Page`) σταθερού μεγέθους (`PAGE_SIZE`), διασφαλίζοντας memory alignment.
- **Sparse vs Dense**: Υποστηρίζεται άμεση πρόσβαση (`direct_access`) για πυκνά δεδομένα, ενώ για στήλες με NULLs χρησιμοποιείται bitmap και βοηθητικοί πίνακες (`chunk_prefix_sum`) για γρήγορη προσπέλαση.

Παραγωγή Ενδιάμεσων Αποτελεσμάτων

Κατά την εκτέλεση των joins, τα αποτελέσματα δεν υλοποιούνται αμέσως.

1. Η κλάση `MatchCollector` (`include/construct_intermediate.h`) συλλέγει τα ζεύγη των row IDs που ταιριάζουν (packed σε `uint64_t`).
2. Η συνάρτηση `construct_intermediate_from_columnar` ή `_mixed` διαβάζει τα δεδομένα βάσει των row IDs.
3. Τα δεδομένα προστίθενται στη δομή `column_t` χρησιμοποιώντας βελτιστοποιημένες μεθόδους `append` και `append_consecutive`. Η `append_consecutive` χρησιμοποιεί `memcpy` όταν εντοπίζονται συνεχόμενα row IDs, επιταχύνοντας σημαντικά την εγγραφή.

Υποστήριξη Μικτών Εισόδων (Mixed Inputs)

Το σύστημα σχεδιάστηκε ώστε να είναι ευέλικτο και να υποστηρίζει ζεύξεις μεταξύ διαφορετικών μορφών δεδομένων (`include/construct_intermediate.h`). Υλοποιήσαμε handlers για τρεις περιπτώσεις:
1. **Columnar-to-Columnar:** Και οι δύο είσοδοι είναι αρχικοί πίνακες (raw pages).
2. **Intermediate-to-Intermediate:** Και οι δύο είσοδοι είναι αποτελέσματα προηγούμενων joins (`column_t`).
3. **Mixed:** Η μία είσοδος είναι αρχικός πίνακας και η άλλη ενδιάμεσο αποτέλεσμα. Αυτό αποφεύγει περιττά βήματα materialization ενδιάμεσα στο πλάνο εκτέλεσης.

Υλοποίηση Hash Join

Στο τρίτο μέρος, υλοποιήσαμε έναν **Unchained Hashtable** (πίνακα κατακερματισμού χωρίς αλυσίδες), βελτιστοποιημένο για cache locality. Η υλοποίηση βρίσκεται στο αρχείο `include/hashtable.h`.

Αρχιτεκτονική Structure of Arrays (SoA)

Χρησιμοποιούμε τρία παράλληλα διανύσματα αντί για array of structs:

1. **Directory (`directory`)**: Ένας πίνακας δεικτών που οδηγεί στα δεδομένα. Κάθε εγγραφή είναι 64-bit και περιέχει κωδικοποιημένη πληροφορία (offset και Bloom filter).
2. **Keys (`keys`)**: Ένας συνεχόμενος πίνακας που αποθηκεύει τα κλειδιά (`int32_t`).
3. **Row IDs (`row_ids`)**: Ένας παράλληλος συνεχόμενος πίνακας που αποθηκεύει τα IDs των γραμμών (`uint32_t`).

Αυτή η διάταξη εξασφαλίζει ότι τα δεδομένα είναι συμπαγή στη μνήμη, επιτρέποντας στον επεξεργαστή να φορτώνει αποδοτικά τις γραμμές cache (cache lines) κατά τη διάρκεια του probe.

Ενσωμάτωση Bloom Filters

Για την επιτάχυνση των αποτυχημένων αναζητήσεων (early rejection), ενσωματώσαμε ένα **Bloom Filter** απευθείας μέσα στο directory. Όπως φαίνεται στη συνάρτηση `finalize_directory` του `hashtable.h`, κάθε 64-bit εγγραφή του directory χωρίζεται ως εξής:

- **Upper 48 bits**: Δείκτης (offset) στο διάνυσμα των `keys`/`row_ids`, που υποδηλώνει το τέλος του bucket.

- **Lower 16 bits**: Ένα μικρό Bloom filter για το συγκεκριμένο bucket.

```
// include/hashtable.h
inline void finalize_directory(const std::vector<uint32_t> &final_offsets) {
    for (size_t i = 0; i < directory.size(); ++i) {
        uint64_t end_idx = final_offsets[i];
        uint64_t bloom = directory[i] & 0xFFFF;
        directory[i] = (end_idx << 16) | bloom;
    }
}
```

Κατά το probing (`find_indices`), ελέγχουμε πρώτα το Bloom filter. Αν το κλειδί δεν υπάρχει στο φίλτρο, η συνάρτηση επιστρέφει αμέσως `{0, 0}`, γλιτώνοντας την πρόσβαση στην κύρια μνήμη των κλειδιών.

Αλγόριθμος Κατασκευής και Βελτιστοποιήσεις

Η κατασκευή γίνεται σε δύο περάσματα (Count -> Scatter) για τέλεια χρήση μνήμης χωρίς ανακατανομές. Επιπλέον, υλοποιήσαμε δύο μονοπάτια εκτέλεσης:

1. **Πρώτο Πέρασμα (Count)**: Υπολογίζουμε πόσα στοιχεία αντιστοιχούν σε κάθε bucket (histogram). Στη συνέχεια, υπολογίζουμε το prefix sum αυτών των μετρητών για να βρούμε την ακριβή θέση εκκίνησης κάθε bucket στα arrays keys και row_ids.
2. **Δεύτερο Πέρασμα (Scatter)**: Τοποθετούμε τα στοιχεία στις προ-υπολογισμένες θέσεις τους και ταυτόχρονα ενημερώνουμε το Bloom filter του αντίστοιχου bucket.

Δυναμική Επιλογή Build Side

Στο στάδιο της προετοιμασίας (`include/join_setup.h`), το σύστημα συγκρίνει το πλήθος των γραμμών των δύο πινάκων εισόδου. Επιλέγεται δυναμικά ο **μικρότερος πίνακας** ως Build side. Αυτή η στρατηγική ελαχιστοποιεί το μέγεθος του Hash Table και, κατά συνέπεια, την κατανάλωση μνήμης (L2/L3 cache footprint).

Nested Loop Join

Ως εναλλακτική λύση για περιπτώσεις που δεν ενδείκνυται Hash Join, υλοποιήσαμε τον αλγόριθμο Nested Loop Join (`include/nested_loop.h`). Η υλοποίηση είναι «page-aware», δηλαδή διατρέχει τις σελίδες εξωτερικά και εσωτερικά, και ελέγχει τα bitmaps των sparse σελίδων ώστε να εκτελεί συγκρίσεις μόνο για έγκυρες εγγραφές, αποφεύγοντας άσκοπους κύκλους επεξεργασίας.

Συμπεράσματα

Συνοψίζοντας, στο παρόν παραδοτέο υλοποιήσαμε μια ολοκληρωμένη μετάβαση από row-store σε column-store αρχιτεκτονική, εστιάζοντας στη μεγιστοποίηση της απόδοσης μέσω της αποδοτικής διαχείρισης της μνήμης και της κρυφής μνήμης (cache) του επεξεργαστή.

Οι βασικοί πυλώνες της υλοποίησής μας ήταν:

1. **Late Materialization:** Μέσω της δομής `value_t` και του bit-packing, πετύχαμε δραστική μείωση του memory bandwidth, μεταθέτοντας την ανάκτηση των «βαριών» δεδομένων (strings) μόνο στο τελικό στάδιο της εκτέλεσης.
2. **Column-Store Διαχείριση:** Η εισαγωγή της δομής `column_t` και η σελιδοποιημένη (paged) αποθήκευση των ενδιάμεσων αποτελεσμάτων εξασφάλισαν καλύτερο cache locality και επέτρεψαν την αποδοτική διαχείριση sparse δεδομένων μέσω bitmaps και prefix sums.
3. **Βελτιστοποιημένο Hashing:** Η υλοποίηση του Unchained Hashtable με Structure of Arrays (SoA) διάταξη και ενσωματωμένα Bloom filters ελαχιστοποιεί τα cache misses και επιταχύνει σημαντικά τη διαδικασία του probing στις ζεύξεις hash joins.

Ο συνδυασμός των παραπάνω τεχνικών καθιστά το σύστημα ικανό να διαχειρίζεται αποδοτικά πολύπλοκα πλάνα εκτέλεσης με ελάχιστο overhead.