CW2I: Community Data Indexing for Complex Query Processing

Mei Hui, Panagiotis Karras, and Beng Chin Ooi

National University of Singapore
{huimei, karras, ooibc}@comp.nus.edu.sg

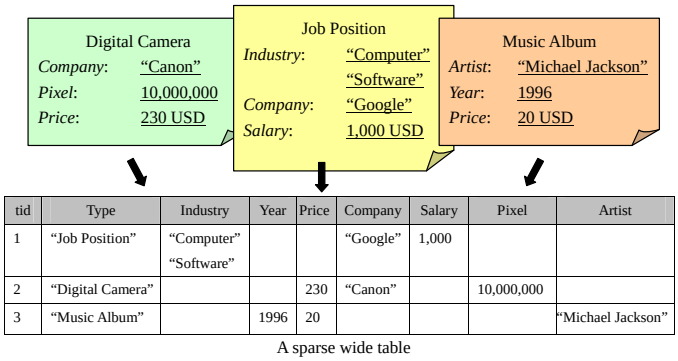
Abstract. The increasing popularity of Community Web Management Systems (CWMSs) calls for tailor-made data management approaches for them. Still, existing CWMSs have mostly focused on simple similarity-based queries; they do not provide a framework for the efficient processing of more complex queries over community web data. In this paper, we propose a two-way indexing scheme that facilitates efficient and scalable retrieval and complex query processing with community data. A thorough experimental comparison, based on real-world data and practical queries, illustrates the advantages of our scheme compared to other approaches for community web data management. Besides, our double-indexing scheme provides an attractive solution to the storage problem as well.

1 Introduction

Community Web Management Systems (CWMSs) are effective in collecting and organizing the wisdom of crowds [13]. They provide a platform in which users of a community can interact and collaborate. Users can contribute to, and take ownership of, the content so as to display their collective knowledge. In general, a CWMS stores information on a wide-ranging set of entities, such as products, commercial offers, or persons. Due to diverse product specifications, user expectations, or personal interests, the data set can be very sparse when rendered as a table. Thus, such systems also raise design requirements that differ from those of conventional database systems.

For instance, the data set of the CNET e-commerce system examined in [10] comprises of a total of 2,984 attributes and 233,304 products; still,a product is described by only ten attributes on average. Likewise, most community-based data publishing systems allow users to define metadata and store information in the way they prefer (see Figure 1). For example, a subset of GoogleBase [1] that we downloaded comprises of 1,147 attributes over 779,019 items, while on average 16 attributes are defined per tuple.

A popular storage structure for CWMSs is the Sparse Wide Table (SWT). Sparse wide tables were independently proclaimed in [18] and [10] as a useful component of self-managing database systems; they free users from a potentially ineffective schema-design phase when managing sparse data sets. Furthermore,



tid	Type	Industry	Year	Price	Company	Salary	Pixel	Artist
1	"Job Position"	"Computer" "Software"			"Google"	1,000		
2	"Digital Camera"			230	"Canon"		10,000,000	
3	"Music Album"		1996	20				"Michael Jackson"

Fig. 1. Submission of user-defined meta data to sparse wide table.

[18] was the first work to propose a type of inverted indexing scheme for sparse table data management.

Nevertheless, most existing CWMSs either are disadvantaged by a lack of scalability to large data sets, or offer good performance only for specialized kinds of queries. None of the solutions has provided to date a scheme that can handle ever-increasing amounts of data as well as allow efficient processing of general-purpose complex queries.

In this paper, we combine two effective indexing methods: First, an *inverted* index for each attribute gathers a list of tuples, sorted by tuple ID, for each attribute value; the inverted index is sorted by the attribute value itself. Second, a separate *direct* index for each attribute provides fast access to those tuples for which the given attribute is defined, sorted by tuple ID, following a column-oriented architecture inspired by [8, 9, 16]. We adopt conventional query optimization techniques to our framework and discuss the benefit gained by *uniting* the inverted indexes of less-frequent attributes. We conduct a performance evaluation using the GoogleBase dataset and compare our proposed method to existing ones. The results confirm that the indexing scheme we propose outperforms the systems based on a monolithic vertical-oriented or horizontal-oriented representation. Our proposed CW2I scheme can efficiently handle complex queries over Community Data.

Outline The remainder of this paper is structured as follows. Section 2 presents related work on SWT storage and discusses the deficiencies of previous approaches. In Section 3 we introduce our two-way indexing solution for SWT data management and a typology of queries defined over such a system. Section 4 presents our experimental results using real-world data sets. Finally, in Section 5 we outline our conclusions.

2 Background and Related Work

It has been long observed that conventional horizontal-schema database representations are not suited for emerging applications with sparsely populated and

rapidly evolving data schemas. In this section we present an overview of existing approaches to handling such data by existing Community Web Management Systems.

2.1 Binary Vertical Representation

A natural approach to handling sparse relational data is to split a sparse horizontal table into as many binary (2-ary) tables as the number of attributes (columns) in the sparse table. This idea was fist suggested in the context of database machines [14] and was brought up again with the decomposition storage model [11, 12]. This type of column-store model has been followed by MonetDB, along with an algebra to hide the decomposition [8, 9], as well as C-Store [16], gaining benefits of compressibility [3] and performance [5]. Furthermore, in a visionary paper [2], Abadi suggested that, apart from data warehouses and OLAP workloads, column-stores may also be well suited for storing extremely sparse and wide tables.

2.2 Ternary Vertical Representation

Agrawal et al. [6] discerned that a ternary (3-ary) vertical representation offers a middle design point between the n-ary horizontal representation of conventional RDBMSs for non-sparse data and the binary vertical representation outlined above. They found that this vertical representation does uniformly outperform the horizontal representation for sparse data, yet the binary representation performs better. In response, [6] suggested the use of multiple, *partial indexes*, i.e., one index on each of the three columns of the ternary vertical table, along the lines of [15]. A premonition of a multiple-indexing approach is also contained in this suggestion.

2.3 Interpreted Storage Format

Beckmann et al. [7] suggests a storage format that attains independence from schema width. Unfortunately, as [7] observed, this renders the retrieval of values from attributes in tuples significantly more complex. To ameliorate this problem, [7] suggests an *extract* operator that returns the offsets to the referenced interpreted attribute values. Still, as [2, 4, 17] have noted, handling sparse tables in such a way incurs a significant performance overhead.

3 CW2I: Two-Way Indexing of Community Web Data

In order to address the aforementioned difficulties, we propose the combination of *three* design approaches for complex query processing on SWTs. *First*, we espouse a binary vertical representation for each attribute *Aid* defined in the data at hand, which collects a *sorted* vector of *Tid* (tuple identifier) entries. Each of these entries is appended with an associated sorted list of attribute

values *Val*. We call this binary representation *direct* index. The binary vertical representation of $[11, 12, 8, 2]$ can be seen as a manifestation of our *direct* index. *Second*, we argue for an inverted index built over each *frequent* attribute. An attribute identifier *Aid* is linked to an *inverted* index, consisting of a sorted vector of *Val* (attribute value) entries, appended with their associated sorted lists of *Tids* $< Tid_1,\ Tid_2,\ \ldots,\ Tid_n >$ that match the given value for the *Aid* attribute. In case an attribute value consists of several short strings, these are independently indexed by our inverted index. String separators prevalent during data entry are used for this purpose. *Third*, we store a single list of attribute-value pairs for each tuple, as in [7]. This design amounts to double indexing scheme for Community Web data, which we call CW2I. It is capable to address both lookup and aggregation queries, as well as more complex queries involving several join operations. In effect, it robustly handles all ways of querying the data.

3.1 The Unified Inverted Index

The main problem arising out of attribute frequency skewness is that, if we are supposed to build both a *direct* and an *inverted* index for each attribute, then we raise unreasonable storage requirements without much of a benefit arising from them. After all, if an attribute is defined for only a few tuples, then not much stands to be gained by indexing the few values it assumes over the whole data set in both a *direct* and an *inverted* manner. Furthermore, most users are not expected to express queries using the names of thousands of lesser-used attributes. Therefore, we suggest that *less frequent attributes* may best be seen as repositories of *unstructured* information about tuples. As far as *text* attributes are concerned, this information can be appropriately considered as a collection of keywords related to the tuples in question. Thus, we propose to handle lesser used *text* attributes in a *unified* manner, gathering all of them together in a *unified inverted index*. This index provides a powerful keyword-search-like functionality over these attributes, while it increases both the storage-efficiency and the user-friendliness of our system. In particular, the unified inverted index gathers a sorted vector of attribute value entries, regardless of the attribute they correspond to; each entry is associated to a list $<$*Tid* (tuple identifier), *Aid* (attribute identifier)$>$ pairs.

It remains to be decided how attributes are to be divided into *more frequent* and *less frequent* ones, so as to decide which should receive their own separate inverted indexes, and which are to be managed by the unified inverted index. After experimentation, we decided that the following rule of thumb provides a reasonable dividing line: The top *quartile* of *text* attributes, sorted by decreasing frequency value, have a separate inverted index built for them. The rest $\frac{3}{4}$ of attributes are to be treated by the unified inverted index.

Numerical attributes are excluded from the unified inverted index; this discussion pertains to *text* attributes only. However, numerical attributes that fall below the frequency threshold to receive their own inverted index, are not indexed in this manner at all. Given the sparsity of such attributes, an inverted index is redundant for them. A lookup to their direct index is sufficient to detect

any tuples that match a given value-based predicate. Besides, such numerical values do not offer anything in terms of keyword-search. Users are expected to refer to such attributes by name. The same reasoning applies to the case when a user needs to refer to a specific lesser-used text attribute by name; again, the low density of the attribute itself renders a lookup for values matches in the direct index practicable enough.

The usage of this unified inverted index addresses a problem that is particular to the community web data we examine. Moreover, it confers the following advantages to our CW2I system:

- Storage efficiency, as it is not efficient from a storage point of view to have a separate inverted index on each of the myriads of less-frequent attributes.
- Facilitation of keyword-search queries, in which a given string is to be found in *any* less frequent attribute. Resorting to a unified index of all less-frequent attributes for keyword-search is more efficient than checking many small indexes. Besides, lesser-used text attributes can be validly seen as collecting keywords related to the specific domain where an entry belongs.
- User-friendliness, users are not expected, or required, to know obscure attribute by name. Users are mostly familiar with the names of the most commonly-used attributes, but, naturally, they cannot easily figure out by what name the others are entered. These lesser-used attributes are usually domain-specific and their appearance depends on the values of other attributes.
- Functionality and practical sense: for rarely used attributes, the direct index is already good enough for a lookup, hence the inverted index can be spared. Thus, while the said benefits are gained by unifying of what could be several smaller indexes, not much is significantly lost.
- Safeguard against user inconsistency. Different users are likely to define the same lesser-used concept with differently-named attributes. Thus, it makes sense to collect the values they provide under one unified index. A keyword search on the values of such attributes is guaranteed to return the tuples related to them (i.e., there are no false negatives).

3.2 Query Typology

As we have discussed, several attribute values in Community Web systems usually appear as collections of short strings. Such strings are usually distinguished by separators. Each user may employ diverse separators (such as '>', '/', ':', ';',) to delimit short strings. Our inverted index distinguishes these short strings and creates a separate entry for each of them. A string match during query processing can be satisfied either in an *exact* or a *fuzzy* manner. Exact string matching is straightforward. For the case of fuzzy matching, we still wish to take advantage of lexicographic order for fast query processing. Thus, we say that a query string s_q of length L and an indexed string s_i satisfy a *fuzzy string match* when there is an *exact* match of the first half of the query string and a similarity between their

other parts. Thus, if $\mathsf{prefix}(K, s)$ is the length-K prefix of string s, we define a fuzzy mach as:

$$\mathsf{prefix}\left(\left\lceil \frac{L}{2} \right\rceil, s_q\right) = \mathsf{prefix}\left(\left\lceil \frac{L}{2} \right\rceil, s_i\right) \wedge \quad \mathsf{suffix}\left(\left\lfloor \frac{L}{2} \right\rfloor, s_q\right) \approx \mathsf{suffix}\left(\left\lfloor \frac{L}{2} \right\rfloor, s_i\right)$$

Where $\approx$ denotes an approximate string similarity measure. According to this kind of fuzzy matching, short strings like 'accessories' would match with 'accessorize' and 'accessory', but not with 'access, windows version'. In the case of text match, we define the fuzzy match score between two text values s_i and s_j as:

$$\text{Score}(s_i, s_j) = N/min(len(s_i), len(s_j))$$

where N is the number of matched words in s_i and s_j, $len(s_i)$ is the number of words in text value s_i and $len(s_j)$ is the number of words in text value s_j. We set a threshold τ, when $\text{Score}(s_i, s_j) \geq \tau$, we say s_i matches s_j. This rule of thumb operates well in practice, allowing for the identification of related strings with tolerance to orthographic and terminological variations.

We distinguish four different types of queries that CW2I can process, based on their exploitation of inverted indexes, unified inverted index, and fuzzy string matching, as follows. A general *complex query*, covering all four types, is defined as follows.

$$Q\{p_1(G_1), p_2(G_2), \ldots, p_n(G_n), r_1(G_{r_1}, G'_{r_1}), \quad r_2(G_{r_2}, G'_{r_2}), \ldots, r_m(G_{r_m}, G'_{r_m})\}$$

where $p_i(G_i)$ is set of (select) predicates that define a group of tuples G_i and $r_i(G_{r_i}, G'_{r_i})$ is a (join) relation among the tuples in groups G_{r_i}, G'_{r_i} which satisfy their respective predicates. For instance, consider the query "find the black-color jewelry and purple-color jewelry such that their price difference is less than 10$". Then G_1 is 'black color jewelry', hence the predicate that defines this group is

$$\mathsf{Product} = \textit{jewelery} \wedge \mathsf{Color} = \textit{black}$$

Likewise, G_2 is 'purple color jewelry', hence the predicate that defines it is

$$\mathsf{Product} = \textit{jewelery} \wedge \mathsf{Color} = \textit{purple}$$

The relation that should be satisfied among any item $t_1 \in G_1$ and any item $t_2 \in G_2$ is

$$|t_1.price - t_2.price| \leq 10\$$$

We can then classify the basic join queries that satisfy this general-purpose definition in to four distinct classes as follows.

1. ***Exact* join query *without* keyword.** In such a query, the select predicates are all on most-frequent, *indexed* (i.e., having their own inverted index) attributes. The join relation operates on a *simple* (i.e., numerical or single-string) attributes (e.g., price, name, but not multiple-short-string or text attributes, on which no exact match can be done).

2. ***Exact* join query *with* keyword.** In this type of query, the select predicates may be defined on published attributes or may be just keyword-specified, referring to less-frequent attributes, lacking their own *inverted index*. The join relation is defined on a simple attributes, as in Type 1.
3. ***Fuzzy join* query *without* keyword.** In this case, the select predicates are on indexed attributes. However, the join operation is a *fuzzy join* on *text* or multiple-short-string attributes. For example, consider the query "Find a cellphone and a laptop of the same brand.". Then G1 is "cellphone", G2 is "laptop", the predicate defines G1 is:

   $$\mathsf{Product} = \textit{cellphone}$$

   Similarly the predicate defines G2 is:

   $$\mathsf{Product} = \textit{laptop}$$

   The join operation on G1 and G2 is:

   $$\textit{t1}.\textit{brand} = \textit{t2}.\textit{brand}$$

   Given that *brand* is a text attribute, the value of *brand* consists of several short strings. Thus the join operation on this attribute should not be done with exact match. We call this type of query *fuzzy join* query.
4. ***Fuzzy* join query *with* keyword.** A query of this type may have both a select predicate defined by a non-indexed attribute, as well as a *fuzzy* join operation on a *text* attribute.

4 Experimental Evaluation

In this section we discuss experimental studies of the scalability and performance of CW2I scheme. We compare the performance of CW2I in answering Type-1 queries with the straightforward horizontal storage scheme (HORIZ) and vertical storage scheme (VERTI). The query processing time is recorded for Type-2, Type-3 and Type-4 queries which are not evaluated in existing systems. We study the performance and the scalability of CW2I on them.

4.1 Experiment Setup

In VERTI, we store each attribute as a separate table and build index on the tuple_id column. This is same as the direct index in CW2I. The difference of implementation between CW2I and VERTI is that we create one inverted index for each of the indexed attributes and a unified index for each of the other attributes. We build a B^+ tree index on the keyword column of the inverted index and the unified index. In HORIZ, we store the indexed attributes in one relational table and the other un-indexed attributes in the vertical storage format. We build a B^+ tree index on TID (tuple id) column. Our experimental environment is a Intel Core2 Duo 1.8GHz machine with 2GB memory, 160GB hard disk, running Windows XP Professional with SP2.

4.2 Description of Data

We downloaded published data items from GoogleBase, and set up our experimental evaluation on this dataset. It consists of 30 thousands tuples which are described by 1319 attributes out of which 1217 are text attributes and others are numerical attributes. According to our statistics, the average number of attributes per tuple is 16. To test the storage cost of the three methods, we initially insert 10k tuples. We measure the additional disk space cost by incrementally inserting 5K tuples each time, as shown in Figure2. We observe that the storage space linearly increases with the number of tuples inserted. CW2I consumes about 25% more disk space than VERTI and HORIZ.

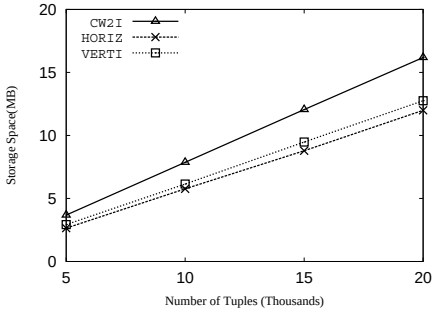


Fig. 2. Disk Space Cost of the Three Methods.

4.3 Description of Queries

We discuss the four types of queries in Section 3.2. In our experiment we generate several queries for each query type. For Type-1 queries, i.e. exact join query without keyword, we compare the execution time of CW2I to VERTI and HORIZ. We describe the queries and the implementation details below.

Type-1 Queries We first outline the Type-1 queries we have included in our experimental study.

- **Query 1.** Find the number of products that belong to the mp3 type.
- **Query 2.** Find the stores which sale both mp3 and computer products.
- **Query 3.** Find pairs of an mp3 and a headphone of the same color.
- **Query 4.** Find a black jewelry item and a purple jewelry item such that the difference of their price is less than 20$.
- **Query 5.** Find a keyboard and a mouse of the same condition(new, used, on sale etc.).
- **Query 6.** Find pairs of books and CDs such that their condition is *new* and they are both sold at the same price.

Type-2 Queries We now define the Type-2 queries we have used for our experimental evaluation.

- **Query 1.** Find the stores which sell both an mp3 and a computer with 250GB disk.
- **Query 2.** Find pairs of two 'ipods', such that one's price is less than 200$, the other's price is more than 200$, and the difference of their prices is less than 100$.
- **Query 3.** Find the stores that sale both Mahal's CD and Dorina's CD.
- **Query 4.** Find thinkpad T41s and thinkpad T20s that the difference of their prices is less than 100$.

Type-3 Queries We have also defined two Type-3 queries, as follows.

- **Query 1.** Find brands that make both cellphones and laptops.
- **Query 2.** Find types of products such that there exists both at least one *red* item and at least one *blue* item of that product_type.

Type-4 Queries Lastly, we have also included two queries of the most complex type in our typology, Type-4, in our study, which are outlined below.

- **Query1.** Find *hardcover* books and Mahal's CDs with the same allowed form of payment.
- **Query2.** Find products of material 'stone' and products of material 'silver', having the same product_type.

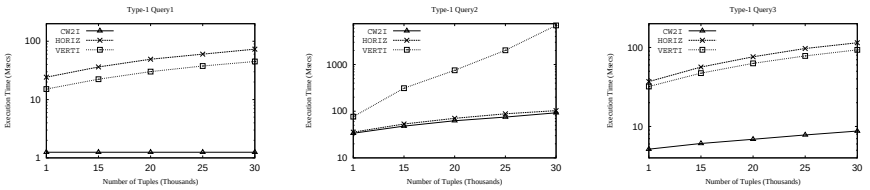
4.4 Results

In this section we report the results of our experimental study with the queries described in Section 4.3. We use progressively larger prefixes of the experimental data set while measuring the execution time of each query. We use logarithmic y-axes for the execution time in each case. Typically, the performance of our CW2I scheme is 10 or more times better than HORIZ and VERTI.

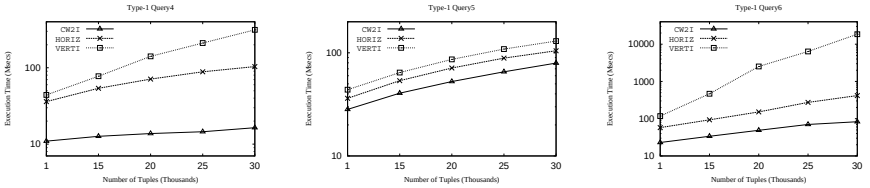
Figure 3(a) shows the execution time results for Type-1 Query 1. In this case, CW2I largely outperforms both our prototype implementation of HORIZ and VERTI. Thanks to the availability of an inverted index in CW2I, only one B^+-tree search is required in order to find the number of qualifying items. In contrast, a whole table scan is necessitated by both HORIZ and VERTI.

Given that the size of the vertical table is smaller than that of the horizontal table, the performance of VERTI is noticeably better than that of HORIZ. Besides, the growth of execution time with the size of the data set is perceptible for both HORIZ and VERTI; on the other hand, the execution time remains relatively stable for CW2I, as new relevant items are inserted in the idlist of the inverted index and can still be easily retrieved without substantial overhead.

Our results for Query 2 of Type-1 is shown in Figure 3(b). We observe that both CW2I and HORIZ outperform VERTI, while CW2I is slightly better than



(a) Execution time, Type-1 Query1. (b) Execution time, Type-1 Query2. (c) Execution time, Type-1 Query3.

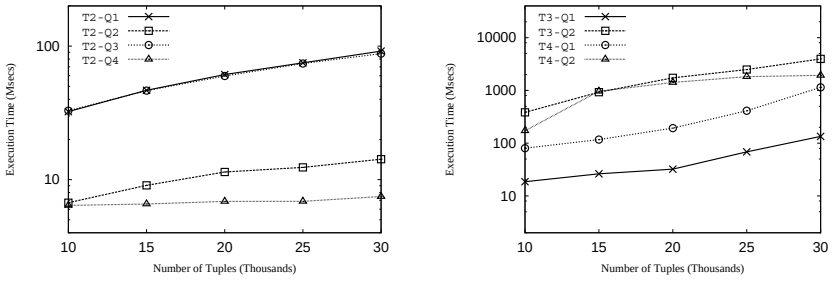


(d) Execution time, Type-1 Query4. (e) Execution time, Type-1 Query5. (f) Execution time, Type-1 Query6.

Fig. 3. Execution time of Type-1 Queries

HORIZ itself. Thanks to the inverted index built on attribute product_type, CW2I gets the tuple_ids of mp3 and computer with little cost. On the other hand, VERTI has to scan the vertical table of attribute product_type twice for that purpose. As far as HORIZ is concerned, a self join on attribute store_id is conducted quite efficiently in this case.

Figure 3(c) depicts our results on Type-1 Query3. Observably, CW2I gains a significant advantage over both HORIZ and VERTI. The underlying cause of this efficiency advantage is the same as in our preceding analysis for Query 2. However, now this advantage is more perceptible within the examined data set sizes.



(a) Execution time, Type-2. (b) Execution time, Type-3,4.

Fig. 4. Execution time of Type-2,3,4 Queries

Our results for Type-1, Query4 are illustrated in Figure 3(d). In this case, the efficiency benefit of CW2I over the other two schemes is remarkable. The reason for the advantage by CW2I remains the same as we have already analyzed in the case of Type1-Query2. However, in this case, there are two select predicates on items, instead of only one, therefore the gain becomes more noteworthy.

Likewise, Figure 3(e) shows our execution time results with the fifth query in Type 1. The advantage gap on behalf of CW2I is again due to the efficient execution of the select predicate. The cost for CW2I is again lower, even though in this case the growth is more perceptible, due to the characteristic of the data set on the specified select predicates on product_type. Still, the gain grows with the data set size in this case as well.

Figure 3(f) depicts our results for Type-1 Query6. The additional select predicates in this query reduce the number of selected results, therefore the cost of the merge step is reduced. Still, the advantages of CW2I are still salient, exactly thanks to the fact that these limited results are much more easily derived than with the other schemes. Besides, the subsequent join operation is now less demanding.

Figure 4(a) illustrates the results of the Type-2 queries. It is indicated that CW2I scales well for the queries of Type-2 with the increase of the dataset size. Q1 and Q3 are almost identical and they both join on attribute store_id, while Q2 and Q4 join on attribute price. The efficiency gap is due to the fact that the cardinality of attribute price is smaller than attribute store_id, so the second group gets better scalability. Moreover, the difference between the results of Q2 and Q4 is due to the difference of the sizes of the intermediate results.

Figure 4(b) shows our results for queries of Type-3 and Type-4. CW2I scales well for the queries of these two types. Still, compared to Type-2 queries (Figure 4(a)), those of Type-3 and Type-4 are less scalable. This difference is due to the fact that the join operations of Type-3 and Type-4 are fuzzy joins, which cost more than the exact join operations of Type-2. Besides, we observe differences among the queries of Type-3 and Type-4 themselves, which are due to variations in result set sizes.

5 Conclusions

In this paper we have proposed an architecture for the management of sparse and wide data in Community Web systems that can efficiently handle complex queries. Our proposal combines two hitherto distinct approaches. On the one hand, we employ a vertical representation scheme, whereby each attribute, no matter how frequently defined, obtains its own column-oriented *direct* index. On the other hand, we utilize an *inverted* indexing scheme, whereby the tuples that define *more frequent* attributes are indexed *by value*. Furthermore, we propose a *unified inverted* indexing scheme that gathers together all *less frequent* in a single keyword-oriented index. This additional index facilitates schema-agnostic keyword search that fits the nature of such less frequent attributes. We have defined four distinct types of join queries that our CW2I system can naturally process.

Our experimental study confirms that CW2I enables fast and scalable processing of complex queries over Community Data more efficiently than systems based on a monolithic vertical-oriented or horizontal-oriented representation, and gains an advantage of several orders of magnitude over them in our prototype implementation.

References

1. Google base. Online at: http://base.google.com/.
2. D. J. Abadi. Column stores for wide and sparse data. In *CIDR*, 2007.
3. D. J. Abadi, S. R. Madden, and M. Ferreira. Integrating compression and execution in column-oriented database systems. In *SIGMOD*, 2006.
4. D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach. Scalable semantic web data management using vertical partitioning. In *VLDB*, 2007.
5. D. J. Abadi, D. S. Myers, D. J. DeWitt, and S. Madden. Materialization strategies in a column-oriented DBMS. In *ICDE*, 2007.
6. R. Agrawal, A. Somani, and Y. Xu. Storage and querying of e-commerce data. In *VLDB*, 2001.
7. J. L. Beckmann, A. Halverson, R. Krishnamurthy, and J. F. Naughton. Extending RDBMSs to support sparse datasets using an interpreted attribute storage format. In *ICDE*, 2006.
8. P. A. Boncz and M. L. Kersten. MIL primitives for querying a fragmented world. *VLDB Jounral*, 8(2):101–119, 1999.
9. P. A. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-pipelining query execution. In *CIDR*, 2005.
10. E. Chu, J. Beckmann, and J. Naughton. The case for a wide-table approach to manage sparse relational data sets. In *SIGMOD*, 2007.
11. G. P. Copeland and S. N. Khoshafian. A decomposition storage model. In *SIGMOD*, 1985.
12. S. Khoshafian, G. P. Copeland, T. Jagodis, H. Boral, and P. Valduriez. A query processing strategy for the decomposed storage model. In *ICDE*, 1987.
13. B. Li, M. Hui, J. Li, and H. Gao. iVA-File: Efficiently indexing sparse wide tables in community systems. In *ICDE*, 2009.
14. M. Missikoff. A domain based internal schema for relational database machines. In *SIGMOD*, 1982.
15. M. Stonebraker. The case for partial indexes. *SIGMOD Rec.*, 18(4):4–11, 1989.
16. M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. R. Madden, E. J. O'Neil, P. E. O'Neil, A. Rasin, N. Tran, and S. B. Zdonik. C-store: a column-oriented DBMS. In *VLDB*, 2005.
17. C. Weiss, P. Karras, and A. Bernstein. Hexastore: Sextuple indexing for semantic web data management. *Proc. VLDB Endowment*, 1(1):1008–1019, 2008.
18. B. Yu, G. Li, B. C. Ooi, and L.-Z. Zhou. One table stores all: Enabling painless free-and-easy data publishing and sharing. In *CIDR*, 2007.