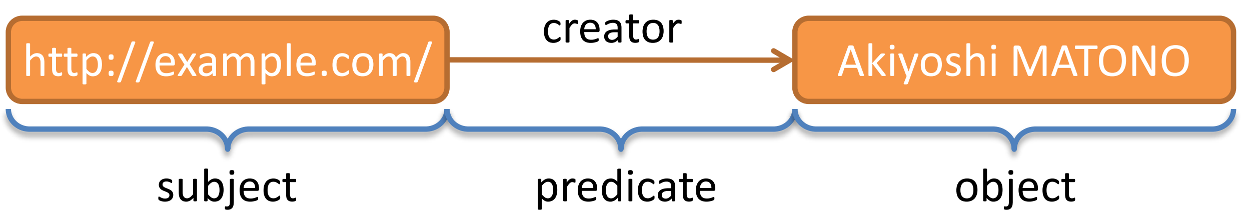
A Storing Scheme and A Merge Join Algorithm for RDF Query Processing

Akiyoshi MATONO
National Institute of Advanced Industrial Science and Technology(AIST)
Japan

Contents

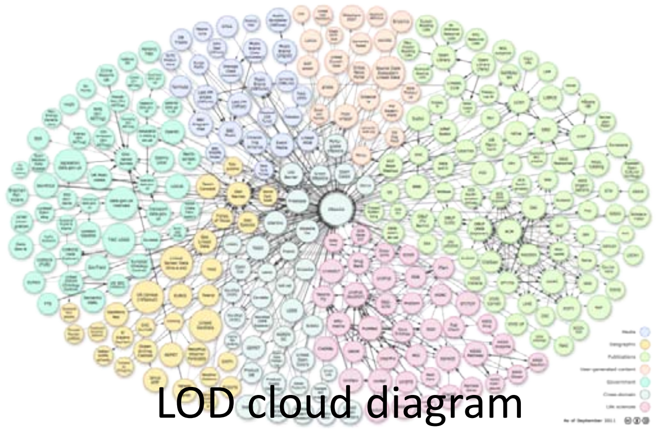
Background

- RDF (Resource Description Framework)
  - RDF is proposed for realizing the Semantic Web vision.
  - RDF is flexible and concise model for representing metadata of resources.
  - RDF data consists of a set of RDF triples.
  - A statement of a resource is described by an RDF triple.
  - A triple is composed of a subject, a predicate, and an object.



Motivation

- Importance of RDF databases
  - RDF data is increasing rapidly
    - e.g.) Linked Open Data had grown to 31 billion RDF triples (Sep 11')
    - Efficient search is an essential issue.
  - RDF query is complex.
    - The structure of RDF data is a directed graph.
      - RDF query is equal to extract sub-graphs from an RDF graph.
    - SPARQL
      - a standard for RDF query language
      - a syntactically-SQL-like language for querying RDF graphs via pattern matching


LOD cloud diagram

Two Approaches

- ✓ Paragraph Table
  - A storing scheme of RDF data into relational tables.
  - RDF data is stored after performing some join operations to reduce the number of the join operations when query processing.
  - Paragraph Table determines which joins should be connected before storing them based on the structure of given RDF documents.
- Bloom Filter Merge Join
  - A merge join algorithm for low-selectivity
    - In RDF data, low-selectivity join operations are frequently used.
  - BFMJ can reduce disk I/O cost by skipping unnecessary parts.
  - BFMJ traverses two B+ trees which comparing two nodes on them to check whether the sets of the descendant keys are disjoint.
  - We extend bloom filter to be suitable for disjoint test which is a test whether two sets contain intersection or not.
  - We extend B+ tree to make each internal node possess the extended bloom filters to represent its descendant keys.

Classification of RDF databases

- Many RDF databases have been proposed.
  - e.g. Jena, Sesame, RDF-3X, Virtuoso, D2R
- RDB is used as its back-end storage systems.
- Storage schemes can be classified into three based on the structure of the back-end layout.
  - Triple store
  - Vertical partitioning
  - Property table

RDF storage scheme: triple store

Triple store

Subject	Predicate	Object
id:codd	type	FullProfessor
id:codd	name	Codd
id:codd	teach	id:course1
id:course1	name	AAA
id:course1	room	0123
id:codd	teach	id:course2
id:course2	name	BBB
id:course2	room	0124
id:codd	email	codd@...
id:jim	type	AssistantProfessor
id:xxx	type	Paper
id:xxx	name	XXX
id:xxx	author	id:codd
id:yyy	type	Book
id:yyy	name	YYY
id:yyy	author	id:codd
id:yyy	author	id:jim

- Schema layout
  - The simplest storage scheme
  - One relational table with three columns
  - Each triple is stored into each row.
- Features
  - Many self-join operations are required to construct answers.
  - Join ordering cannot apply to it because it cannot estimate the statistics of each predicate.
  - The table becomes so huge that the performance of selection operation also declines.

RDF storage scheme: vertical partitioning

Vertical partitioning

resource	type
id:codd	FullProfessor
id:jim	AssistantProfessor
id:xxx	Paper
id:yyy	Book

resource	teach
id:codd	id:course1
id:codd	id:course2

resource	room
id:course1	0123
id:course2	0124

resource	email
id:codd	codd@...

resource	name
id:codd	Codd
id:course1	AAA
id:course2	BBB
id:xxx	XXX
id:yyy	YYY

resource	author
id:xxx	id:codd
id:yyy	id:codd
id:yyy	id:jim

- Schema layout
  - Consists of a set of two-column tables
    - 1st column contains the subjects
    - 2nd column contains the objects
  - Each table is created for each predicate
    - # tables = # kinds of predicates
- Features
  - It can maintain the statistics about predicates.
    - The join ordering can be used.
    - The query performance is better than that of triple store.
  - It also decomposes RDF data into RDF triples.
    - # joins required in a query is the same as that of triple store.

RDF storage scheme: property table

Property table

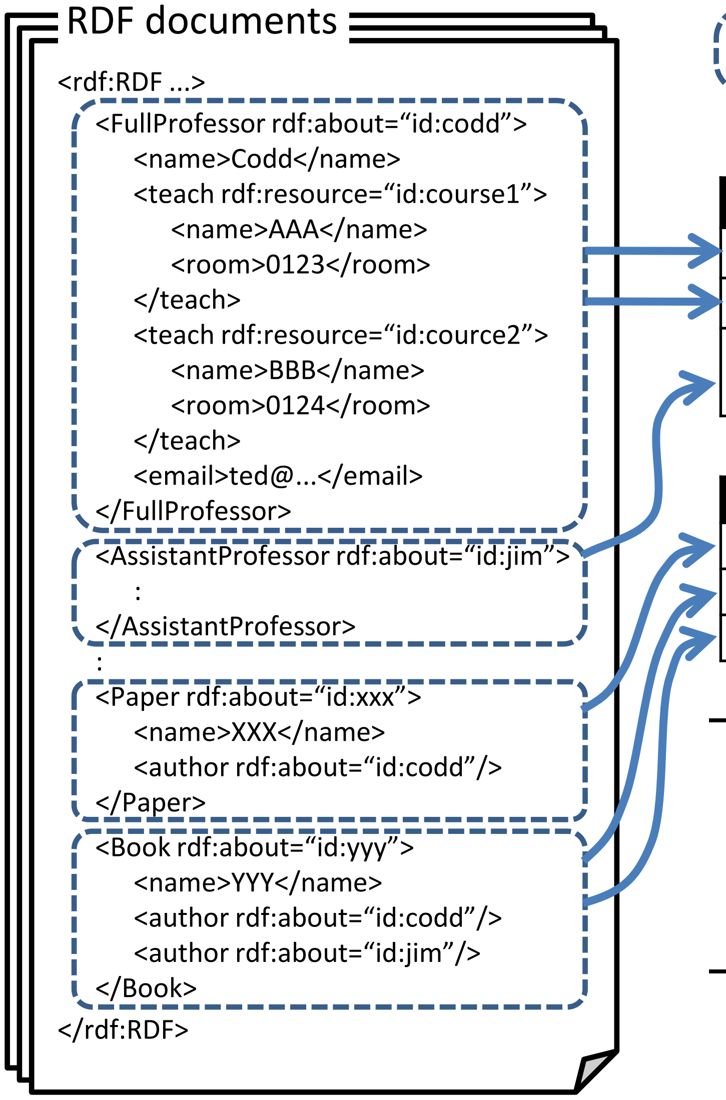
resource	type	name	teach	email
id:codd	FullProfessor	Codd	{id:course1, id:course2}	codd@...
id:jim	AssistantProfessor	...	...	...

resource	name	room
id:course1	AAA	0123
id:course2	BBB	0124

resource	type	name	author
id:xxx	Paper	XXX	{id:codd}
id:yyy	Book	YYY	{id:codd, id:jim}

- Schema layout
  - It's composed of a set of multiple column tables
  - A tuple in a table consists of a set of adjacent triples.
  - Some join operations have been performed before storing them.
- Features
  - It can reduce # of join operations
  - There are some variations
    - In a typical approach, all triples which have the same subject are stored into a tuple.
    - Our proposed paragraph table is one of the property table approaches.

Concept of Our Proposal



RDF documents

```
<rdf:RDF ...>
  <FullProfessor rdf:about="id:codd">
      <name>Codd</name>
      <teach rdf:resource="id:course1">
          <name>AAA</name>
          <room>0123</room>
      </teach>
      <teach rdf:resource="id:cource2">
          <name>BBB</name>
          <room>0124</room>
      </teach>
      <email>ted@...</email>
  </FullProfessor>
  <AssistantProfessor rdf:about="id:jim">
      :
  </AssistantProfessor>
  :
  <Paper rdf:about="id:xxx">
      <name>XXX</name>
      <author rdf:about="id:codd"/>
  </Paper>
  <Book rdf:about="id:yyy">
      <name>YYY</name>
      <author rdf:about="id:codd"/>
      <author rdf:about="id:jim"/>
  </Book>
</rdf:RDF>
```

: RDF paragraph

paragraph table: t1

c1	c2	c3	c4	c5	c6	c7
id:codd	FullProfessor	Ted	id:course1	AAA	0123	ted@...
id:codd	FullProfessor	Ted	id:course2	BBB	0124	ted@...
id:jim	Assistant Professor	...	...	...	...	...

paragraph table: t2

c1	c2	c3	c4
id:xxx	Paper	XXX	id:codd
id:yyy	Book	YYY	id:codd
id:yyy	Book	YYY	id:jim

– RDF document
  - is described in structured language, such as RDF/XML, n-triples and turtle, etc.
  - consists of a set of fragments (**RDF paragraph**)

– Paragraph table
  - stores RDF paragraphs into their corresponding relational tables as they are, without decomposition and connection.

Paragraph Table

Paragraph table: t1

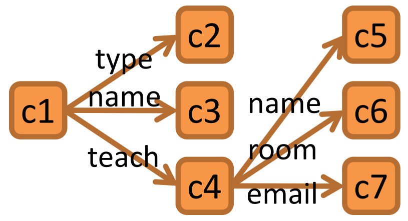
c1	c2	c3	c4	c5	c6	c7
id:codd	FullProfessor	Ted	id:course1	AAA	0123	ted@...
id:codd	FullProfessor	Ted	id:course2	BBB	0124	ted@...
id:jim	Assistant Professor	...	...	...	...	...

The values in these columns may be joined.

Paragraph table: t2

c1	c2	c3	c4
id:xxx	Paper	XXX	id:codd
id:yyy	Book	YYY	id:codd
id:yyy	Book	YYY	id:jim

The structure of the paragraphs in this table



Schema table

tid	subject	predicate	object
t1	c1	type	c2
t1	c1	name	c3
t1	c1	teach	c4
t1	c4	name	c5
t1	c4	room	c6
t1	c1	email	c7
t2	c1	type	c2
t2	c1	name	c3
t2	c1	author	c4

Connection table

tid1	cid1	tid2	cid2
t1	c1	t2	c4
t2	c4	t1	c1
t1	c1	t1	c1
t1	c4	t1	c4
t2	c1	t2	c1
t2	c4	t2	c4

- Schema layout
  - Not only paragraph tables but also a schema table and a connection table.
  - Schema table is used to map the structure of RDF paragraph to table columns.
  - Connection table stores the information of columns in which their values can be connected, because materialization of values which are possible to join to other values are not yet completely finished.
- Feature
  - Our approach is classified as a property table.
  - It can also reduce # join operations.
  - We believe that the storing structure resembles the query structure.
    - RDF paragraph is structured to be easily understood by human.
    - The query is structured in human-readable too.

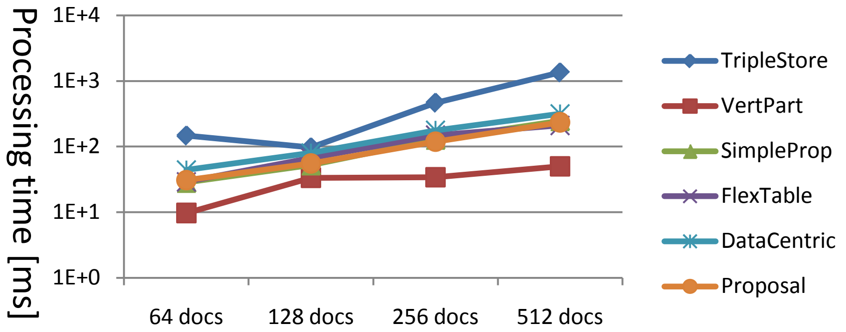
Experimental Evaluation

- Environment
  - CPU: Intel Core2 Quad Q9450 (2.66 GHz)
  - main memory size: 4 Gbytes
  - OS: Ubuntu Linux 10.04 (kernel 2.6.32)
  - HDD: 500BG, SATA, 7200rpm, cache=16MB, seek time=14ms, transfer rate=1546Mbps, latency =4.17ms
- Dataset & Query
  - LUBM benchmark(64, 128, 256, 512 docs)
    - An RDF document contains 8512 triples and its data size is 68 KByte.
  - We used query 1, 2, 4, 7, 8, 9, and 14, because the other queries are designed for inference query.
- Approaches
  - Triple store
  - Vertical partitioning
  - Property table(Simple, FlexTable[1], DataCentric[2], and Proposal)

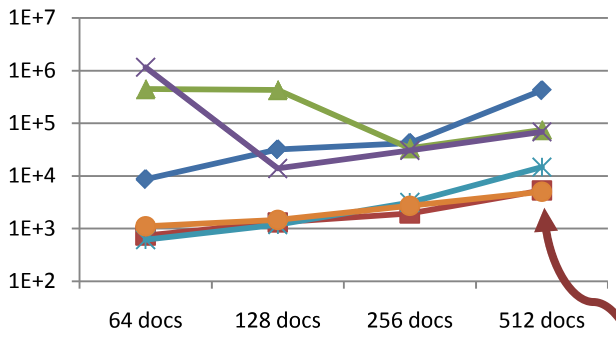
[1] Wang, Y et. al.: Using a dynamic relation model to store RDF data. DASFAA 2010

[2] Levandoski, J.J. et. al.: RDF data-centric storage. ICWS 2009

Results

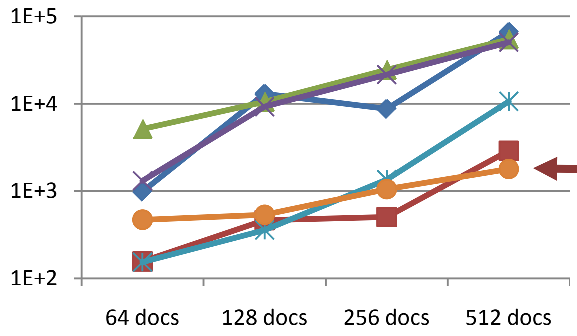


Query 1

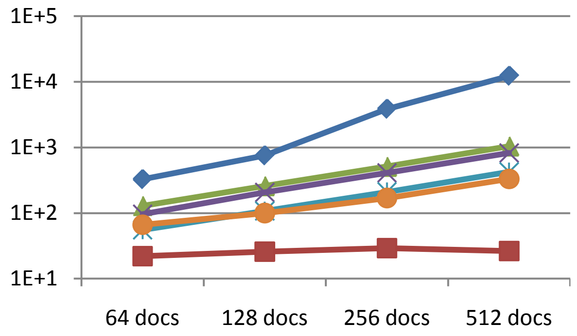


The sum of the times of all queries

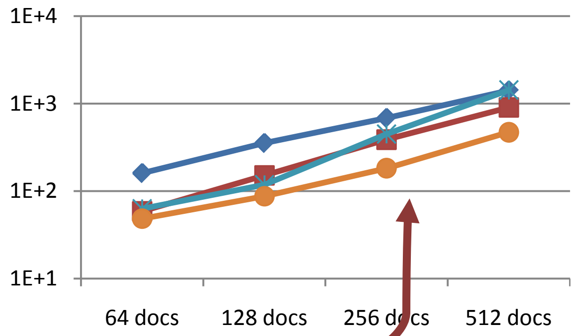
Our approach is the fastest when 512 documents.



Query 2

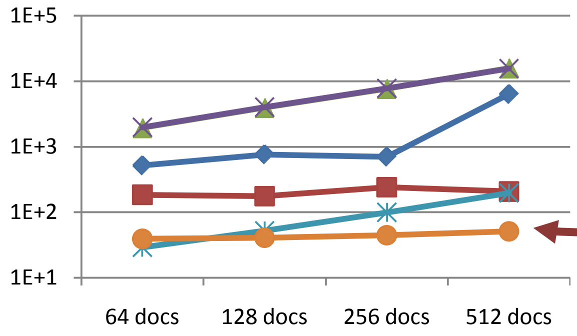


Query 7

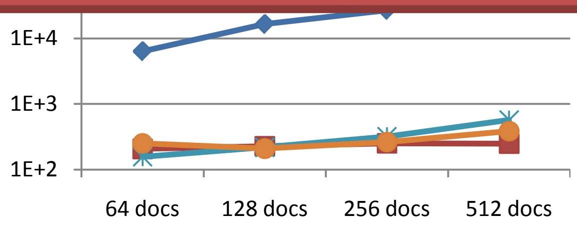


Query 9

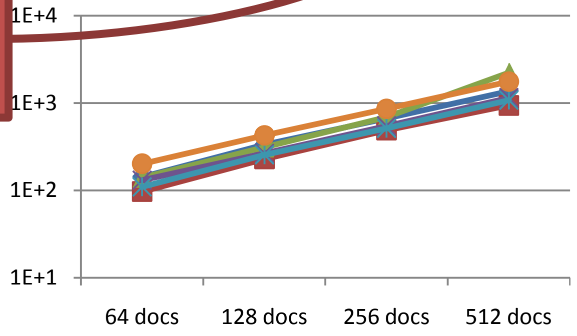
When some queries, our approach overcomes the other approaches.



Query 4



Query 8



Query 14

Summary of Paragraph Table

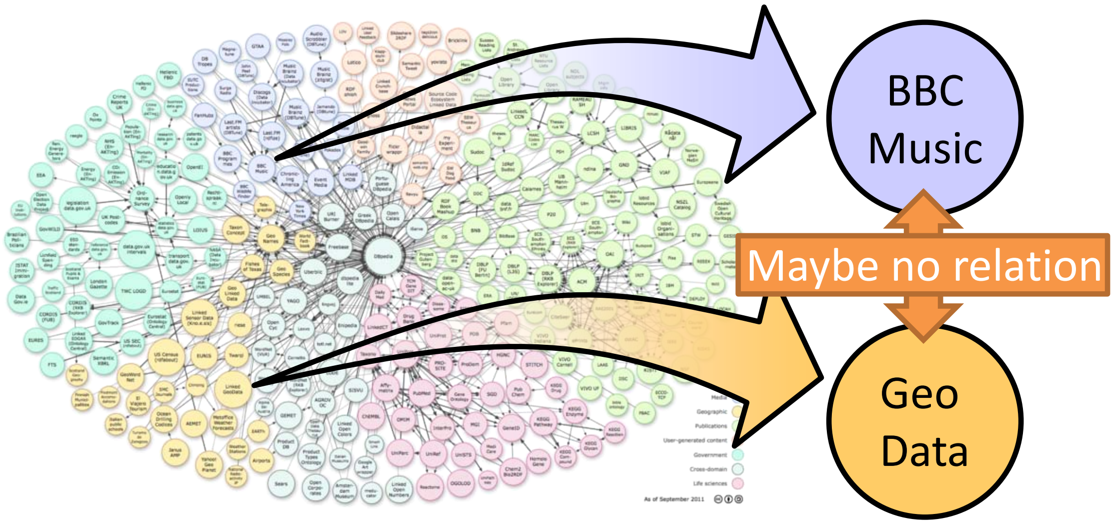
- We proposed an RDF storing scheme (Paragraph Table)
  - The concept of this approach is based on that the structure of input data resembles that of queries.
  - Paragraph Table stores RDF paragraphs into their corresponding relational tables as they are without decomposing or connection.
- We evaluated our approaches through some experiments
  - In the summation of the processing times of all queries, our approach is the fastest when the number of documents is the largest.
  - I think I have to do more experiments using other benchmarks.

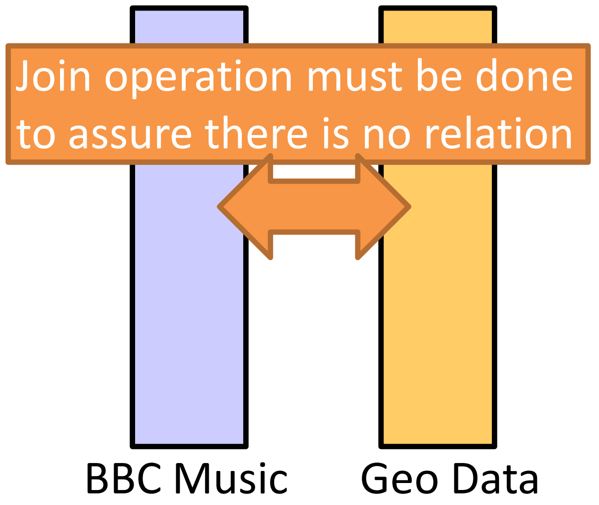
Two Approaches

- Paragraph Table
  - A storing scheme of RDF data into relational tables.
  - RDF data is stored after performing some join operations to reduce the number of the join operations when query processing.
  - Paragraph Table determines which joins should be performed based on the structure of given RDF documents.
- ✓ Bloom Filter Merge Join
  - A merge join algorithm for low-selectivity
    - In RDF data, join operations for low-selectivity are frequently used.
  - BFMJ can reduce I/O cost by skipping unnecessary parts.
  - We extend bloom filter to be suitable for disjoint test which is a test whether two sets contain intersection or not.
  - We extend B+ tree to make each internal node possess the extended bloom filters to represent its descendant keys.
  - BFMJ traverses the two extended B+ trees while comparing two nodes on them to check whether the sets of the descendant keys are disjoint.

Low-selectivity Join in RDF

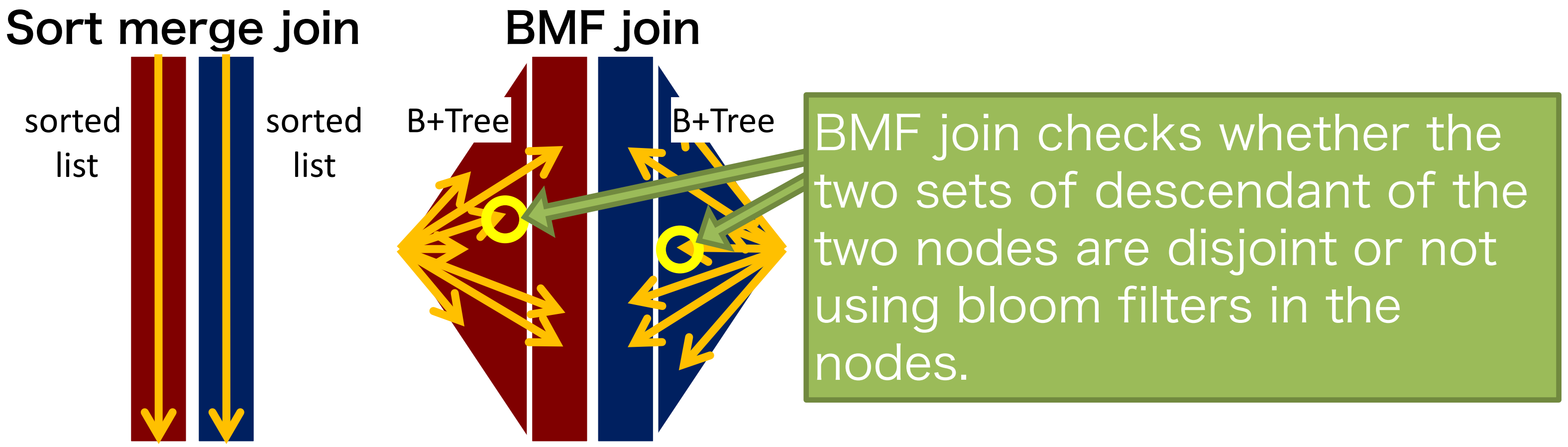
- In RDF, join operations for low-selectivity are frequently used.
  - RDF uses global IDs to identify resources.
  - If resources in different projects are assigned to the same ID, they must be the same resource.
  - Even if two projects may have little relationship, we must perform join operation between them, because there may be the same resources.





Concept of Bloom Filter Merge Join

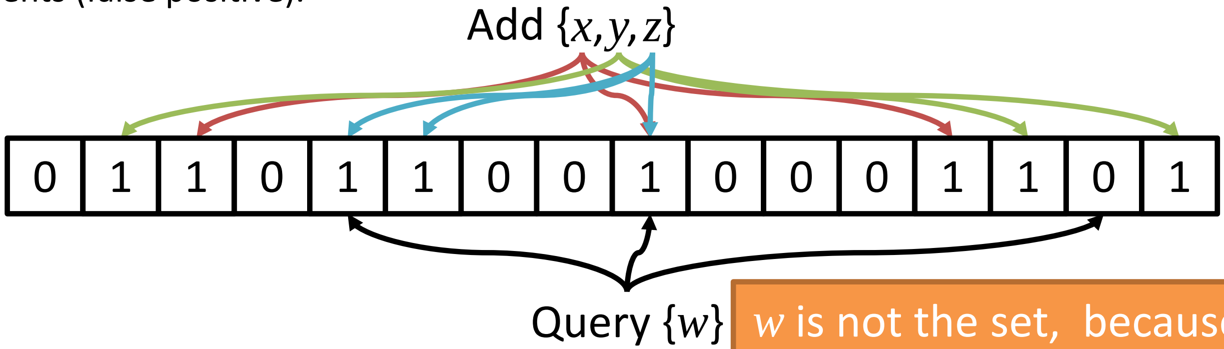
- Sort merge join scans both sorted list, so extra unnecessary disk I/O traffic is caused when a low-selectivity join.
- Bloom filter merge join can skip unnecessary blanches in two B+ trees, in order to reduce disk I/O cost,


Sort merge join
sorted list
sorted list
BMF join
B+Tree
B+Tree
BMF join checks whether the two sets of descendant of the two nodes are disjoint or not using bloom filters in the nodes.

- We propose an **extended B+ tree**, each node of which has not only index information but alto bloom filters to be used for disjoint test between descendant keys.
- We propose an **extended bloom filter** that can be used to check disjoint between two sets.

Bloom Filter

- Features
  - A space-efficient probabilistic data structure that is used to test whether an element is a member of a set.
  - False positives are possible, but false negatives are not
- Preparation
  - a bit array of m bits. All set to 0, if the bloom filter is empty.
  - k different hash functions, each of which maps element to one of the m array positions
- To add an element
  - feed it to each of the k hash functions to get k array positions.
  - set the bits at all these positions to 1.
- To query for an element (IOW, test whether it is in a set)
  - feed it to each of the k hash functions to get k array positions.
  - If any of the bits at these positions are 0, the element is **not** in the set.
  - If all are 1, then either the element is in the set, or the bits have been set to 1 during the insertion of other elements (false positive).

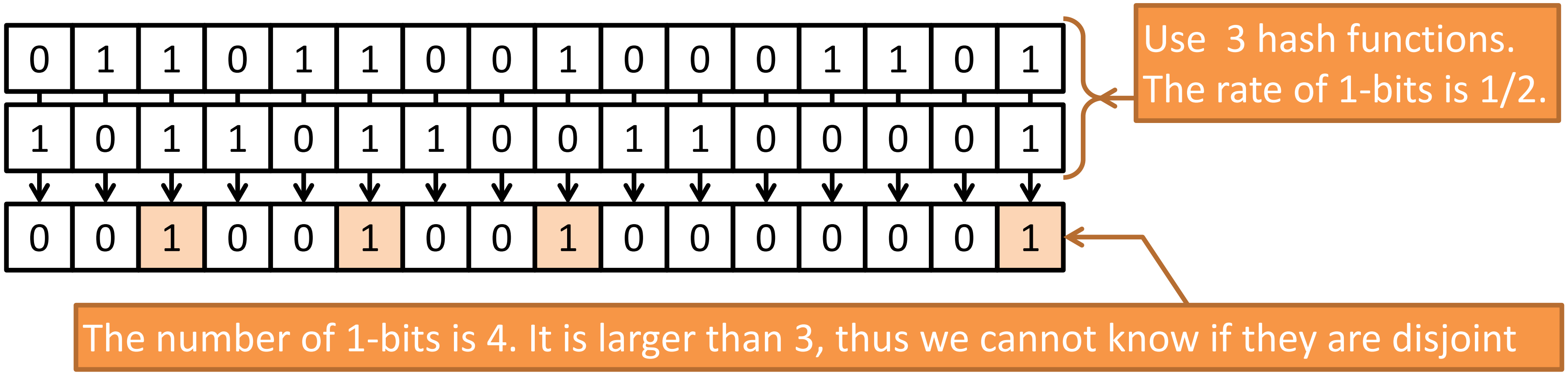


w is not the set, because one position is 0.

Disjoint Test using Bloom Filters

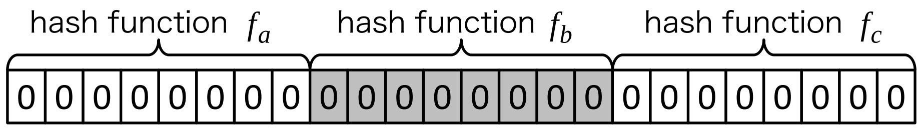
To test if 2 sets are disjoint or not using only bloom filters without the original data

- To test if 2 sets are disjoint using original bloom filters
  1. Perform bit AND operation between the two bit arrays.
  2. If the bit array of the result contains 1-bits less than k, then it assures that the 2 sets are disjoint.
- In general, parameters of a bloom filter is determined to fulfill that the probability that a bit is 1 is 1/2.
  - Because of the space-efficiency.
- After the bit AND operation, the result's probability that a bit is 1 is 1/4.
- There are few cases in which the number of 1-bits is less than k.
- Therefore, the original bloom filters cannot be used for disjoint test.

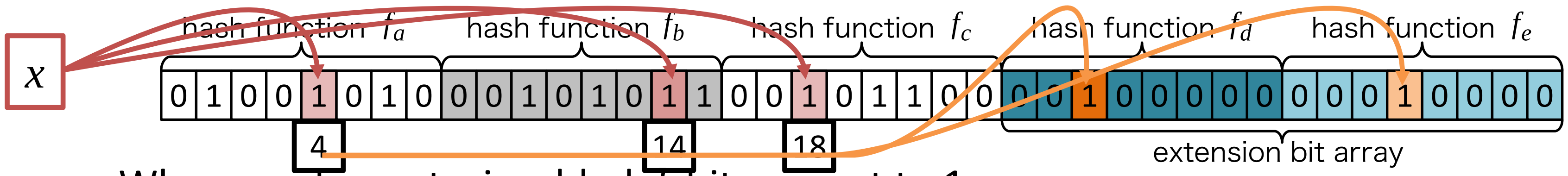

0 1 1 0 1 1 0 0 1 0 0 0 1 1 0 1
1 0 1 1 0 1 1 0 0 1 1 0 0 0 0 1
0 0 1 0 0 1 0 0 1 0 0 0 0 0 0 1
Use 3 hash functions.
The rate of 1-bits is 1/2.
The number of 1-bits is 4. It is larger than 3, thus we cannot know if they are disjoint

Extended Bloom Filter

- Divide a bit array into k sub bit arrays.



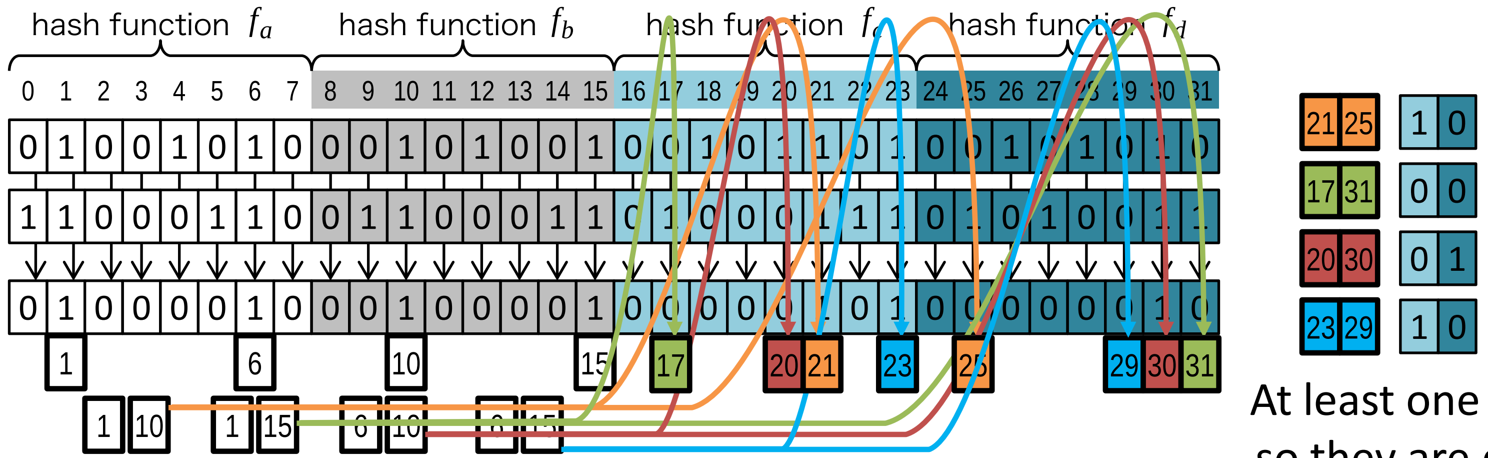
- Add an extension bit array.



  - When an element x is added, k bits are set to 1.
    - 3 bits {4, 14, 18} are set to 1.
  - We don't know whether the 3 bits are set to 1 by an element using the original bit array only.
  - The extension bit array is used to assure that the 3 bits is set to 1 by an element.
  - We prepare functions f_d and f_e which input the positions of the 3 bits and output the positions in the extension bit array.
    - f_d ({4, 14, 18}) = 25, f_e ({4, 14, 18}) = 35
  - Then we set the bits of the results of the functions.

Disjoint Test using Extended Bloom Filter

- Perform bit AND operation between 2 extended bloom filters.
- Get the positions of 1-bits in the front part.
  - {1, 6} and {10, 15}
- Generate all possible combinations from the positions.
  - (1,10), (1, 15), (6,10), and (6, 15)
- Input each combination into the rear functions f_c and f_d
  - (21, 25), (17, 31), (20, 30), and (23, 29)
- Check whether both bits of each pair are set to 1.
  - If there exists any pair which both bits are set to 1, then they may be not disjoint.
  - If, for all pairs, at least one bit is set to 0, then they must be disjoint.

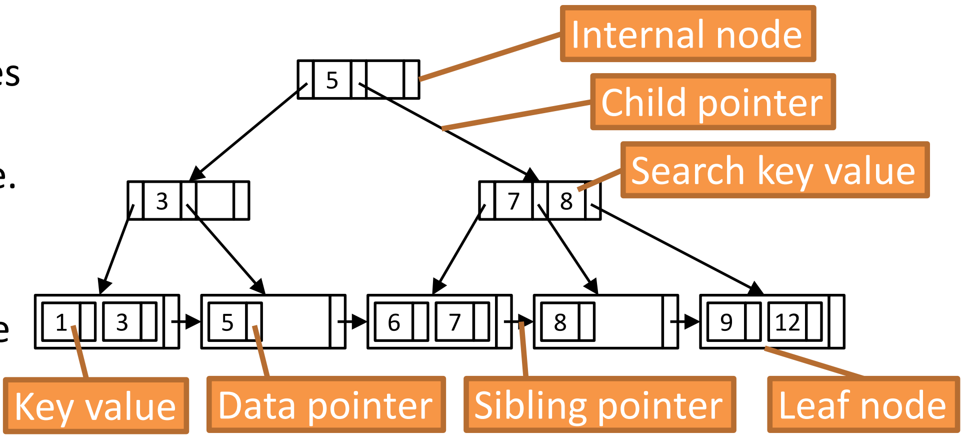


At least one bit is 0, so they are disjoint.

Extended B+ Tree

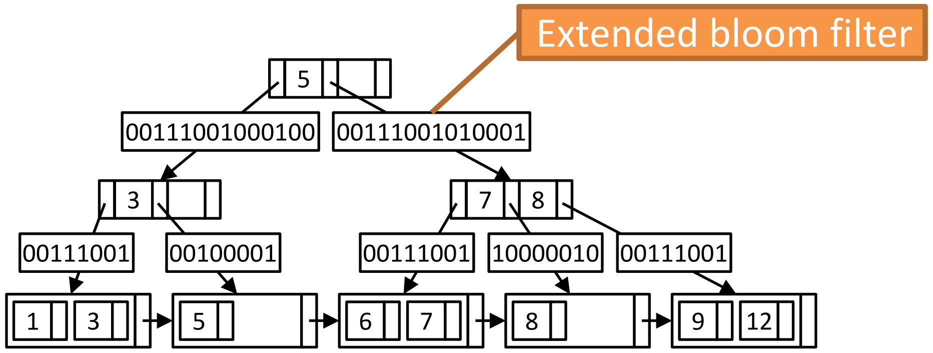
Original B+ Tree

- B+ tree consists of 2 type nodes; Internal nodes and Leaf nodes.
- Internal nodes point to other nodes in the tree.
- Leaf nodes point to data using data pointers.
- Leaf nodes also contain a sibling pointer.
- Both nodes contain keys that are used to guide the search for entries in the index.

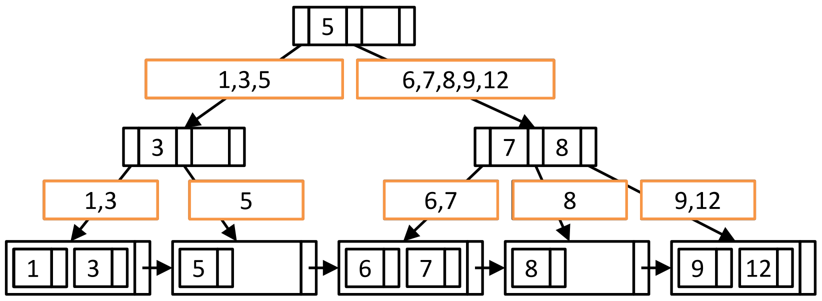


Extended B+ Tree (Bloom B+ Tree)

- Internal nodes contains our extended bloom filters, each of which represents a set of the descendant keys.
- Using the bloom filters, we can infer absence of a key before accessing leaf nodes.

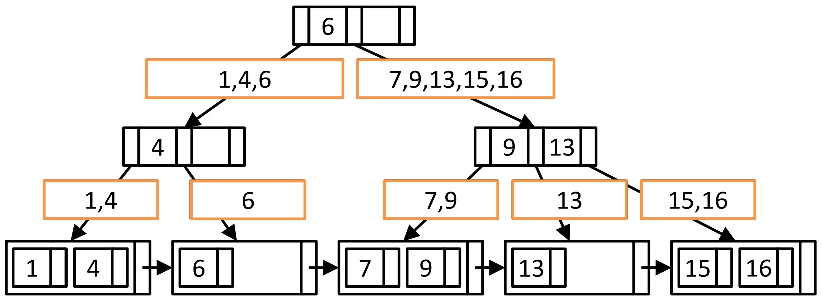


Bloom Filter Merge Join



Not be accessed

The smaller node is moved to the sibling of the ancestor



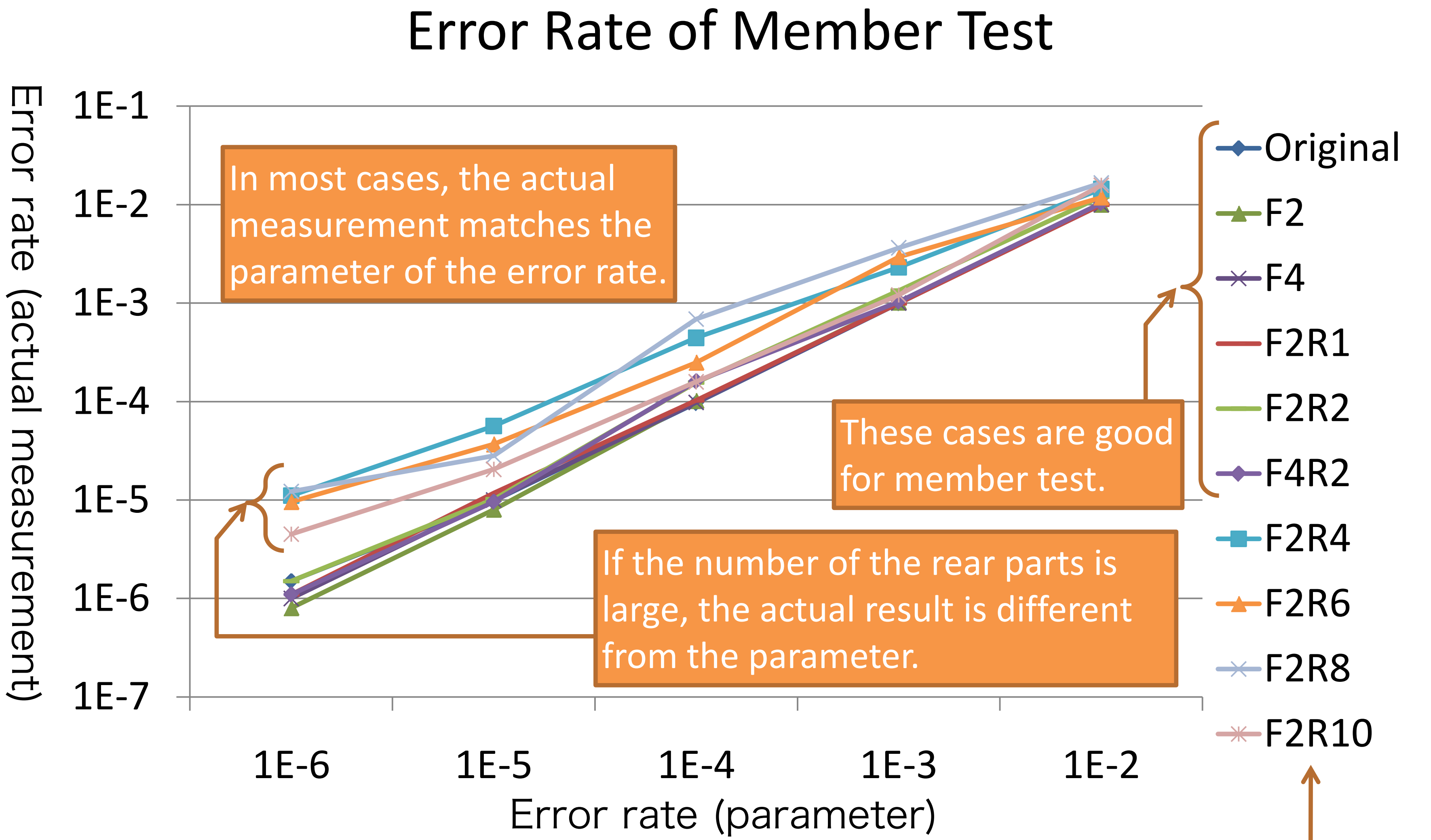
- Traverse B+ trees while comparing between the bloom filters of internal nodes.
- If there is an intersection, then the focuses are moved to their child nodes.
- If there is an intersection and the child nodes are leaf nodes, then their data can be joined.
- If there is no intersection, the focus of the smaller node is moved to the sibling node of the ancestor node.
- Thus, the descendant nodes of the smaller node are never accessed.

Therefore, many nodes can be skipped if the join selectivity is very low.
We can reduce the disk I/O cost to read unnecessary nodes.

Experimental Evaluation

- Experiments
  - Performance evaluation for the extended bloom filter
    - Error rate of member test using a set and non members, which is primary usage of a bloom filter.
    - Error rate of disjoint test using two disjoint sets.
  - Performance evaluation for the bloom filter merge join
    - Processing time for various probabilities of join selectivity.
    - Processing time for various error rates of member test of bloom filters.
- Experimental environment
  - Machine：CPU: Intel Core2 Quad Q9450 (2.66 GHz), main memory size: 4 Gbytes, OS: Ubuntu Linux 10.04 (kernel 2.6.32)
  - Storage：SSD: Intel X25-M Mainstream
  - Implement：using an open source B+ tree (JDBM) for Java

Experimental Results of Bloom Filters



In most cases, the actual measurement matches the parameter of the error rate.

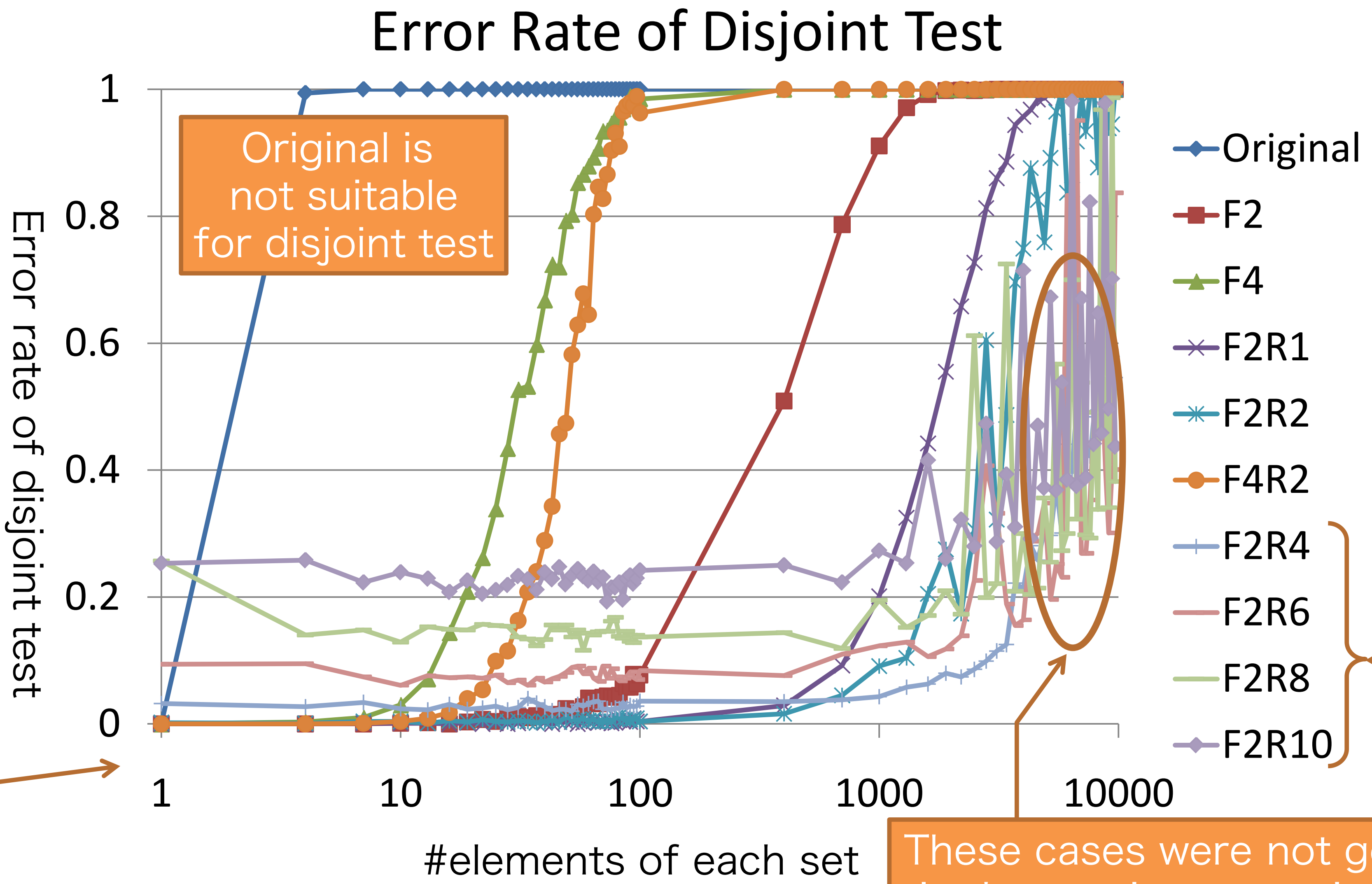
These cases are good for member test.

If the number of the rear parts is large, the actual result is different from the parameter.

F*x*R*y* = a bit array is divided into *x* front parts and *y* rear parts.
Original = the original bloom filter

Experimental Results of Bloom Filters

The lower the error rate is, the better the case is.

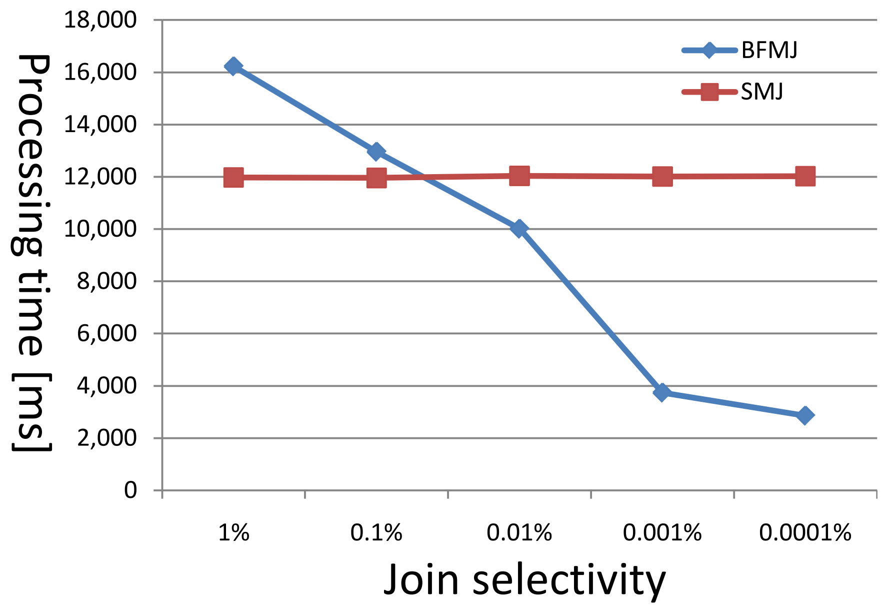


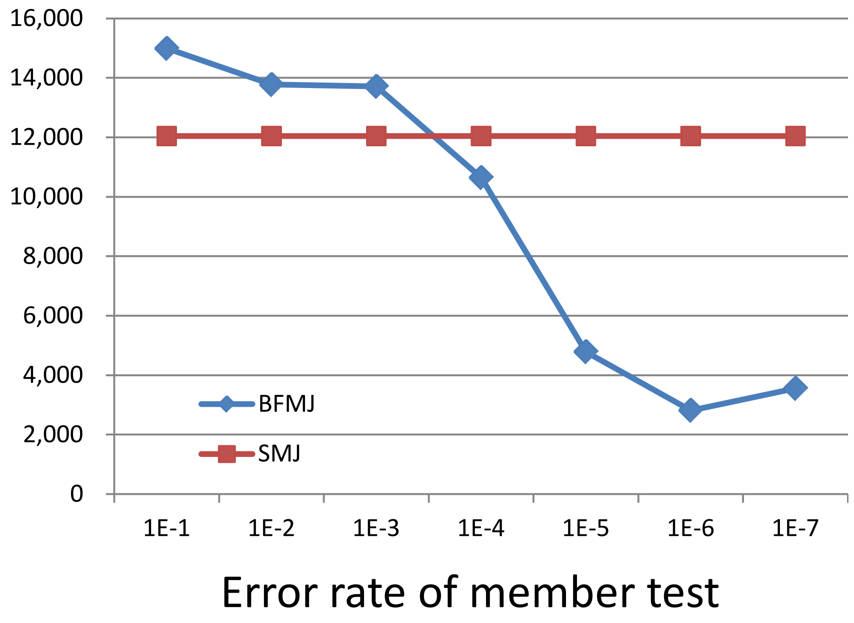
Original is not suitable for disjoint test

These cases were not good in the previous experiment

F2R2 is the best performance

Experimental Results of Bloom Filter Merge Join





- Compare between our bloom filter merge join(BFMJ) and sort merge join(SMJ).
- The processing time of SMJ is always constant.
- When the join selectivity and the error rate of the member test are low, our approach overcomes the sort merge join.

Summery of Bloom Filter Merge Join

- We proposed a merge join algorithm for low-selectivity join query.
  - We proposed an extended bloom filter for disjoint test.
  - We extended B+ tree to make each internal node possess the extended bloom filters in order to represent its descendant keys.
  - Our proposed join algorithm can traverse the extended B+ trees while comparing two nodes in order to check whether an intersection exists in the descendant keys.
- We evaluated our approaches through some experiments
  - Our extended bloom filter can be used for disjoint test.
  - Our proposed merge join is better than the sort merge join when the join-selectivity is low, because our approach can reduce the disk I/O cost by skipping unnecessary parts of the B+ trees.

Conclusion

- We proposed 2 approaches for RDF query processing
  - Paragraph Table is an RDF storing scheme that is based on the structure of RDF documents.
  - Bloom filter merge join is a merge join algorithm that is suitable when the join selectivity is low.
- Future works
  - We will extend the Paragraph Table to apply it to a distributed and parallel environment.
  - We will apply the technology of the extended bloom filter to a hash join algorithm in MapReduce in order to reduce the amount of transfer data.
  - We want to integrate the technologies of paragraph table and the bloom filter merge join.