1 E2017120501: Data Partitioning on Fragmented XML Data

This report describes the experiments used to evaluate the performance of applying data partitioning strategy on fragmented XML data.

1.1 A Brief Introduction to Implementation

The current implementation cannot process all of the target queries (in Table 7) on fragmented XML documents with data partitioning in a distributed-memory environment. There are still problems existed to be solved, which cause some of the queries unable to be successfully evaluated.

1.2 Settings

Hardware There were 1 master (HaoDesk) and 16 workers (matsu-lab00 – matsu-lab15) used in the experiments listed in Appendix A. The master and workers were connected with a local network at speed of 1 Gbps.

Software The implementation is written in Java (64-Bit Server VM version 1.8.0_151). The input XML data set was xmark600 (see Table 5), which was fragmented into 16 fragments with maxisze=4M. On each worker, there ran a BaseX server (Verion: 8.6.7). For each fragment, a BaseX database instance, namely 'xmark600_16_4M' was created in memory on each worker. To understand the memory consumption, the sizes of `xmark600_16_4M` of workers are listed in Table 1.2. In the table, *input size* refers to the size of input XML document; *database size* refers to the size of the database created in BaseX, including all indexes and related data. For the data partition, the number of partions(P) were 1, 2, 3, 4, 8. The intermediate database for holding the results of prefix query were named '`xmark600_16_4M_tmp`'. Since all workers equip with 8 GB memory, we can use set the database in main memory mode.

XQuery Expressions Two XPath expressions XQ1 and XQ2 for prefix and suffix queries are listed in Section C.2.

1.3 Confirming Correctness

The number of hit nodes (as well as the order) and the result sizes have been checked and confirmed to be correct. All the successfully evaluated queries have the same number of hit nodes in original order. Some may be different in size, such as xm3a.dps. Its original result size was 922,270,281, but in this experiment, it is 883,253,777. This dramatical difference is caused by line-break. The original queries were evaluated on Windows, while the current with data partitions on Linux. Since BaseX using '\r\n' on Windows, while '\n' on Linux for line-break, there is one byte difference for each new line. We also found that the query has 6,502,751 hit nodes and there are six lines in every hit node, such

Tab. 1: Information of database sizes on workers.

worker	input size	database size
matsu-lab00	3897 MB	4293 MB
matsu-lab01	3613 MB	3807 MB
matsu-lab02	3662 MB	3788 MB
matsu-lab03	4001 MB	4434 MB
matsu-lab04	3750 MB	4112 MB
matsu-lab05	3907 MB	4255 MB
matsu-lab06	4369 MB	5021 MB
matsu-lab07	4524 MB	5226 MB
matsu-lab08	3605 MB	3876 MB
matsu-lab09	4159 MB	4611 MB
matsu-lab10	3571 MB	3720 MB
matsu-lab11	4005 MB	4434 MB
matsu-lab12	4333 MB	4995 MB
matsu-lab13	4309 MB	4864 MB
matsu-lab14	4219 MB	4743 MB
matsu-lab15	4621 MB	5363 MB

Tab. 2: A hit node of xm3.dps

```
<bidder>
<date>08/04/1999</date>
<time>11:15:36</time>
<personref person="person17793"/>
<increase>7.50</increase>
</bidder>
```

as an example result shown in Table 2.

We then have 883,253,777 + 6,502,751 * 6 - 922,270,281 = 2. These two bytes are extra '\n'. So the sizes are the same.

1.4 Discussion on Queries

The execution time of queries and related information are listed in Table 3.

1.5 Timing

There were three period of times recorded, T_p, T_s and T_m. T_p is the time for executing prefix query. It starts from sending prefix query to all workers and ends with the complete of prefix query on all workers. T_s is the time for executing suffix query, starting from sending till all queries on workers done. T_m is the time for merging results. For comparison, T_o that represents the execution time of xm1.org – xm6.org is also listed in the table.

Tab. 3: Execution time with related information

Key	T_p (s)	T_s (s)					T_m (s)	T_o (s)	Results Size	
		P=1	P=2	P=3	P=4	P=8			hit nodes(M)	in MiB
xm1.dps								3122	5.5	57267
xm2.dps								0.01	0	0
xm3.dps	0.3	113	58.5	30.1	20.2	13.5	6	63.3	6.5	880
xm4.dps								101	0.6	1511
xm5.dps	0.3	243	67.5	48.9	37.8	31.8	10	74.9	35.9	944
xm6.dps								70	1.3	1289

xm1.dps Since the result of xm1.dps is larger than the memory of HaoDesk, this query was not evaluated. This is a design choice about how to process the results. One proposal is to stored the intermediate results in files and then concatenate these files to form the final result.

xm2.dps This query was not tested because this query has no hit node in XMark600 as explained in experiment E20170401. To keep using this query, one proposal is to make a minimal change to the attribute part of the query, such as changing `category52` to `category324329`, which exists in XMark600.

xm3.dps The execution times of suffix part increases with respect to the number of P. Although the workers have only 4 cores, still we can get less execution time with more partitions.

xm4.dps Failed to pass the execution (explained in Section 1.6).

xm5.dps The query xm5a.dps was the target query in the previous design. However, due to the insufficient number of nodes in the results of prefix query to be allocated to 16 computers[1], it was changed it to xm5.dps (xm5b.dps was also evaluated, but it took 1480s (P=4), which is too long and thus ignored). The execution time when P=1 takes about 4 times longer then P=2. The execution from P=2 to P=4 are basically linear.

xm6.dps Failed to evaluate it with data partitioning. The reason is the same as xm5a.dps (both `/site/regions` and `/site/regions/*[...]/item` were tested).

[1] Detailed explanation: An error message "database 'xmark600_16_4M_tmp' has no node with PRE value 5." was encountered when executing a suffix query on an intermediate database. The database had the content of `<root><part> ...</part></root>` , where there is only one `part` node. In a suffix query when P = 2, it is then to process "pre value 5" in XQ2, which refers to the second `part` node. However, since there is no second `part` node, the error occurred.

Tab. 4: Execution times of original queries on matsu-lab11

query	size	avg(3-5)	Original
xm1.org	4.0 GiB	68s	3123s
xm3.org	68.8 MiB	2.3s	63s
xm4.org	53.0 MiB	1.8s	101s
xm5.org	74.3 MiB	2.4s	75s
xm6.org	128.0 MiB	4.2s	71s

1.6 Problems

In general there are several problems (or design choices) listed below:

Evaluation cannot be done. This is the primary problem that hinders the implementation to process all queries successfully. The current implementation for some reason cannot successfully finish the evaluation. It went dead many times in the middle of evaluating suffix queries. For example, letting P=4, there are 64 suffix queries processed in parallel. The evaluation stays irresponsible after 50 or more suffix queries have been evaluated. The reason has not been figured out yet.

Memory size is not enough for xm1.dps. The result of xm1.dps is nearly 60 GB, which is larger than the memory of HaoDesk (32 GB). One proposal is to keep the results on disk.

Performance is low. The evaluations for some suffix takes very long time. For example, when setting P = 4, the execution of xm3.dps on the second partition on matsu-lab11 takes 37 seconds (xm3.org takes 63.3s). Since this evaluation only takes on 1/64 (4 partitions X 16 workers = 64 partitions) of the whole input XML document, it still takes over 1/2 of the original execution time. To understand the performance of the original query on a single worker, an extra experiment was also conducted on matsu-lab11 as shown in Table 1.6. From the results, it suggest that the parallel evaluation on a single partition takes too long time than the original queries.

1.7 Efficiency of Parsing Intermediate Results of Suffix Query

A important method that affects the performance of the implementation is `basex.PreValueReceiver.process(InputStream input)` used to parse the received results of suffix query, (i.e. PRE value + content). It takes the results of suffix query and returns an QueryResultsPre instance with a list of PRE values and a list of string content (of the same size). A simple experiment were done to evaluate the parsing speed. In the experiment, it took 465 ms to parse 52,757 KiB data with 704,430 nodes, i.e. it can process 100 MiB data per second, which

Tab. 5: Statistics of XMark datasets

key	# elements	# attributes	# context	total nodes	file size
xmark1	1,666,315	381,878	1,173,732	3,221,925	113.06 MB
xmark600	1,002,327,042	229,871,111	705,824,967	1,938,023,120	66.99 GB

nearly reach the maximum network speed and thus should be sufficient for not being a bottleneck.

A Hardware

A.1 Computers

A.1.1 HaoDesk

CPU: Intel Core (TM) i7 3930K @ 3.2 GHz, 6 cores 12 threads
Memory: 32 GB DDR3 1333 GHz
Disk: 256GB SSD + 4TB HDD
Windows 7 Professional SP1 64bit

A.1.2 matsu-lab00 – matsu-lab09

All 9 PCs have the same configuration.
CPU: Intel(R) Core(TM) i5-2500 @ 3.3 GHz CPU, 4 cores
Memory: 8 GB
System: Linux version 4.9.0-3-amd64 (Debian 6.3.0-18)

A.1.3 matsu-lab10 – matsu-lab15

All 7 PCs have the same configuration.
CPU: Intel(R) Core(TM) i5-760 @ 2.8 GHz CPU, 4 cores
Memory: 8 GB
System: Linux version 4.9.0-3-amd64 (Debian 6.3.0-18)

B DataSets

B.1 XMark Dataset

The XMark datasets are listed in Table 5.

C Queries

C.1 XPath Quereis

The original XPath queries for XMark datasets are listed in Table 6. The data partitioning XPath queries of Table 6 is listed in Table 7.

Tab. 6: Original XPath queries on XMark datasets.

key	query
xm1.org	`/site//*[name(.)="emailaddress" or name(.)="annotation" or name(.)="description"]`
xm2.org	`/site//incategory[./@category="category52"]/parent::item/@id`
xm3.org	`/site//open\_auction/bidder[last()]`
xm4.org	`/site/regions/*/item[./location="United States" and ./quantity > and ./payment="Creditcard" and ./description and ./name]`
xm5.org	`/site/open\_auctions/open\_auction/bidder/increase`
xm6.org	`/site/regions/*[name(.)="africa" or name(.)="asia"]/item/description/parlist/listitem`

Tab. 7: Data partitioning XPath Queries on XMark datasets.

key	query
xm1.dps	pre = `/site//*`, suf = `[name(.)="emailaddress" or name(.)="annotation" or name(.)="description"]`
xm2.dps	pre = `db:attribute("{db}", "category52")`, suf = `/parent::incategory/parent::item/@id`
xm3.dps	pre = `/site/open_auctions/open_auction`, suf = `/bidder[last()]`
xm4.dps	pre = `db:text("{db}", "Creditcard")` suf = `/parent::*:item[parent::*/parent::*:regions /parent::*:site/parent::document-node()] [(*:location = "United States")][0.0 < *:quantity] [*:description][*:name]`
xm5.dps	pre = `/site/open_auctions/open_auction`, suf = `/bidder/increase`
xm5a.dps	pre = `/site/open_auctions`, suf = `/open_auction/bidder/increase`
xm5b.dps	pre = `/site/open_auctions/open_auction/bidder`, suf = `/increase`
xm6.dps	pre = `/site/regions`, suf = `/self::*[name(.)="africa" or name(.)="asia"]/item /description/parlist/listitem`

C.2 XQuery Expressions

The two XQuery expressions are used for processing prefix query and suffix query for data partitioning strategy shown in Table C.2 and Table C.2 respectively. Note that `{eval_prefix}` will be first replaced by `{prefix}` or `db:open('{db}'){prefix}` depending on whether the query is optimized.

Tab. 8: XQuery Expression for XQ1 prefix part

```
// XQ12: for prefix query
let $d := array { {eval_prefix} !
          db:node-pre(.) } return
for $i in 0 to {P} - 1 return
let $q := array:size($d) idiv {P} return
let $r := array:size($d) mod {P} return
let $part_length := if ($i < $r) then $q + 1
else $q return
let $part_begin  := if ($i <= $r) then ($q + 1) * $i
else $q * $i + $r return
insert node element part {
array:subarray($d, $part_begin + 1, $part_length)
} as last into db:open('{tmpdb}')/root
```

Tab. 9: XQuery Expression XQ2 for suffix part

```
// XQ2: for suffix query
declare option output:method '$mode';
declare option output:item-separator '[';

for $pre in db:open('{tmpdb}')/root return
let $node := db:open('{db}'){suffix}
return (db:node-pre($node), $node)

let $part_pre := {p}*2 + 1 return
for $pre in ft:tokenize(db:open-pre('{tmpdb}', $part_pre)) return
for $node in db:open-pre('{db}', xs:integer($pre)){suffix})
return (db:node-pre($node), $node)
```