Parallel, In Situ Indexing for Data-intensive Computing

October 24, 2011

Jinoh Kim, Hasan Abbasi, Luis Chacon,
Ciprian Docan, Scott Klasky, Qing Liu,
Norbert Podhorszki, Arie Shoshani, John Wu

Introduction

- Many scientific applications produce large outputs
  - For example, GTC generates 260 GB data per 120 sec
  - But, a relatively small fraction of the data is interesting, e.g., blobs and clumps in fusion, magnetic nulls in magnetohydrodynamic models
- Challenge:
  - Accessing data on disk is slow
  - Disk is getting slower relative to computing power
- We explore performance impact on parallelism and in situ indexing for large data

2

ADIOS

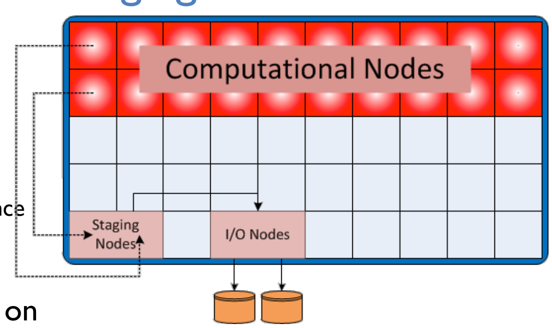
- Adaptable IO Systems developed by ORNL
  - Proven read/write performance
  - Widely adopted as a middleware for data-intensive scientific computing
- Provides good architectural merits for "in situ" processing
  - By decoupling compute nodes with staging nodes
  - Staging nodes take full charges of writing data
- Examples
  - Statistics computation when data is generated
    - Min, max, average, standard deviation

http://www.olcf.ornl.gov/center-projects/adios/

3

Data Staging

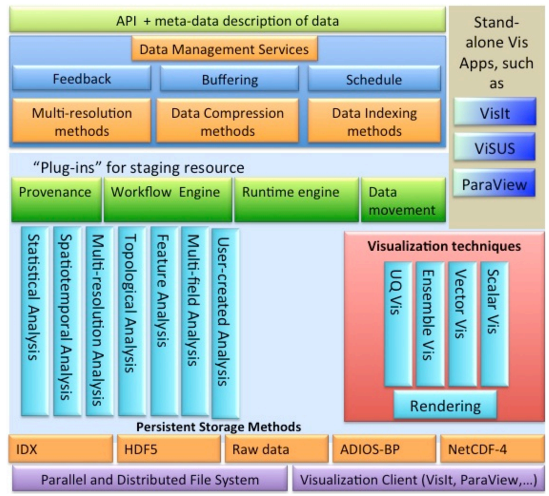
- Why asynchronous I/O?
  - Eliminates performance linkage between I/O subsystem and application
  - Decouples file system performance variations and limitations from application run time



- Enables optimizations based on dynamic number of writers
- High bandwidth data extraction from application
- Scalable data movement with shared resources requires us to manage the transfers
- Scheduling properly can greatly reduce the impact of I/O

In Situ Processing

- The cost of data movement, both from the application to storage and from storage to analysis or visualization, is a deterrent to effective use of the data
- The output costs increase the overall application running time and often forces the user to reduce the total volume of data being produced by outputting data less frequently
- Input costs, especially to visualization, can make up to 80% of the total run time
- Solution: perform analysis operations *in situ* or in place



FastQuery Challenges & Approaches

(1) Mismatch between the array model used by scientific data and the relational model when applying database indexing technology
- Map array data to relational table structure on-the-fly

(2) Arbitrary hierarchical data layout
- Deploy a flexible yet simple variable naming scheme based on regular expression

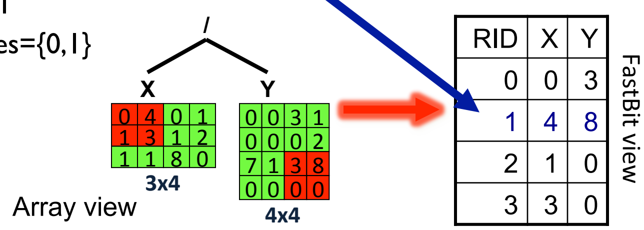
(3) Diverse scientific data format
- Define a unified array I/O interface

(4) High index building cost
- Parallel I/O strategy and system design to reduce the index building time

6

Mapping between FastBit & Array Data

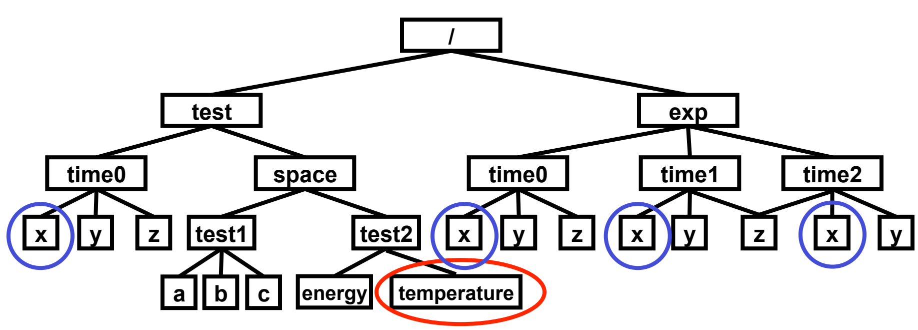
- Each variable associated with a query is mapped to a column of a relational table *on-the-fly*
- Elements of a multidimensional array are linearized
- An arbitrary number of arrays or subarrays can be placed into a logical table as long as they have the same array dimensions
- Ex: getNumHits("x[0:2,0:2] > 3 && y[2:4,2:4]>3")
  - NumHits=1
  - Coordinates={0,1}



7

Flexible Naming Schema

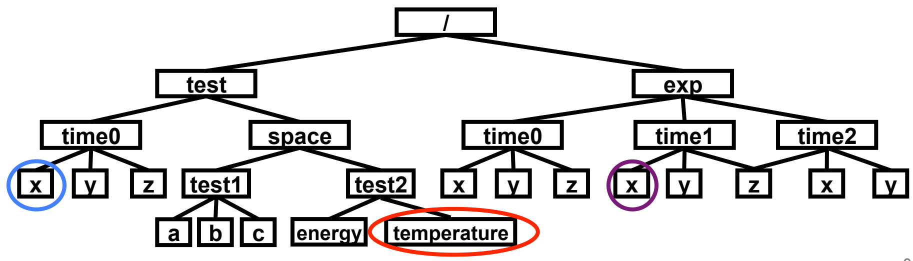
- Naïve option: use the full path
  - getNumHits("/test/space/test2/temperature > 100")
- Can we do better?
  - getNumHits("x > 3")



8

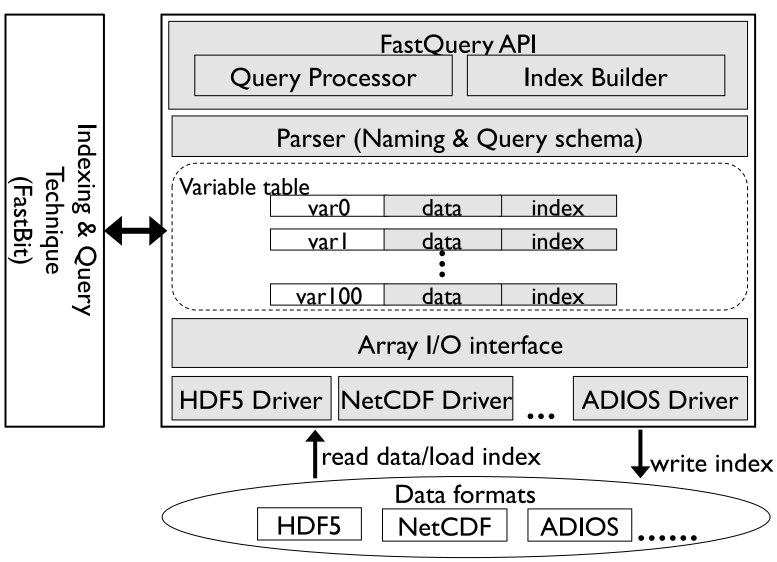
Flexible Naming Schema

- Separate variable name and path
  - Implemented with a tuple (varName, varPath)
  - Variable is identified by the rule "*/varPath/*/varName"
- Example:
  - ("temperature > 100","") ➔ "/test/space/test2/temperature > 100"
  - ("x > 3", test) ➔ "/test/time0/x > 3"
  - ("x > 3", time1) ➔ "/exp/time1/x > 3"
- Advantage:
  - Simplify query string
  - Decouple user specification from file layout



9

FastQuery System Architecture



10

Basic Bitmap Index

Data values	b_0 =0	b_1 =1	b_2 =2	b_3 =3	b_4 =4	b_5 =5
0	1	0	0	0	0	0
1	0	1	0	0	0	0
5	0	0	0	0	0	1
3	0	0	0	1	0	0
1	0	1	0	0	0	0
2	0	0	1	0	0	0
0	1	0	0	0	0	0
4	0	0	0	0	1	0
1	0	1	0	0	0	0

A < 2 (b_0, b_1)   2 < A (b_3, b_4, b_5)

- First commercial version
  - Model 204, P. O'Neil, 1987
- Easy to build: faster than building B-trees
- Efficient for querying: only bitwise logical operations
  - $A < 2 \rightarrow b_0$ OR b_1
  - $A > 2 \rightarrow b_3$ OR b_4 OR b_5
- Efficient for multi-dimensional queries
  - Use bitwise operations to combine the partial results
- Size: one bit per distinct value per row
  - Definition: Cardinality == number of distinct values
  - Compact for low cardinality attributes, say, cardinality < 100
  - Worst case: cardinality = N, number of rows; index size: N*N bits

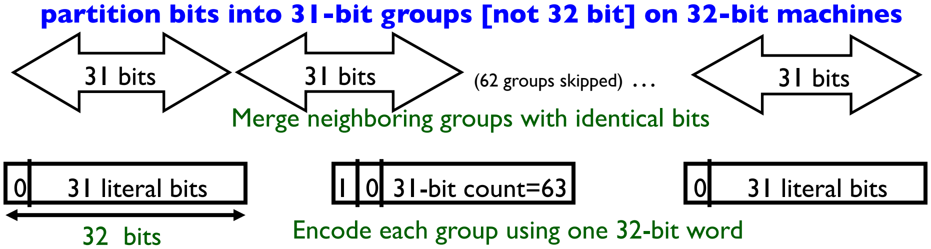
11

FastBit Compression

Example: 2015 bits

100000000000000000000111000000000000000000000000000000000……………..00000000000000000000000000000000111111111111111111111111

Main Idea: Use run-length-encoding, but…
partition bits into 31-bit groups [not 32 bit] on 32-bit machines

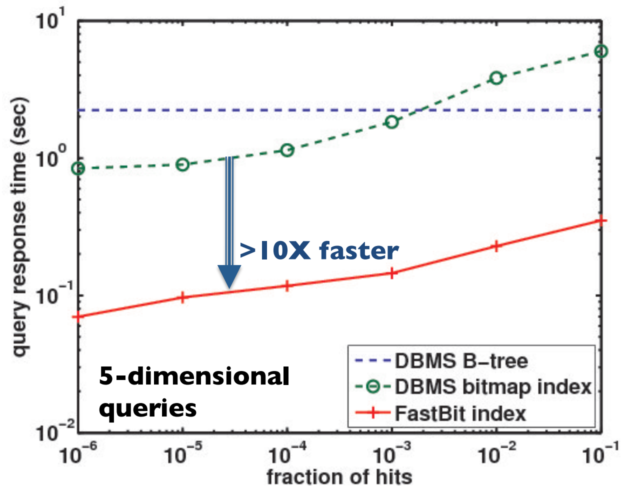


- Name: Word-Aligned Hybrid (WAH) code (US patent)
- Key features: WAH is compute-efficient [Wu, Otoo, and Shoshani 2006]
  - ➢ Uses the run-length encoding (simple)
  - ➢ Allows operations directly on compressed bitmaps
  - ➢ Never breaks any words into smaller pieces during operations
  - ➢ Worst case index size 4N words, not N*N (without compression)

12

Multi-Dimensional Query Performance

- Queries 5 out of 12 most popular variables from STAR (2.2 million records)
- Average attribute cardinality (distinct values): 222,000
- FastBit uses WAH compression
- DBMS uses BBC compression
- FastBit >10X faster than DBMS
- FastBit indexes are 30% of raw data sizes



Wu, Otoo andShoshani 2002

13

Experimental Evaluation

- Impact of indexing
- Parallel index building
- In situ index building

- Measurements collected on Franklin at NERSC
  - ~10000 nodes
  - 8 cores
  - 8 GB memory
  - Lustre file system
- Test problem sizes
  - Small: 3.6GB
  - Medium: 27GB
  - Large: 208GB
  - Large2: 173GB

14

Why Indexing?

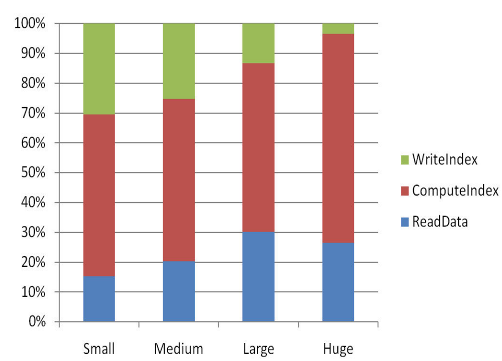
Hits (%)	Method	Small (3.6GB)	Medium (27GB)	Large (208GB)	Huge (1.7TB)
99%	Scanning	38.2s	321.3s	3176.7s	19534
	Indexing	9.6s	32.8s	55.5s	111.8s
	Speed-up	4x	10x	57x	175x
20%	Scanning	37.9s	327.3s	3132.4s	19705
	Indexing	11.7s	61.8s	153.6s	1195.4s
	Speed-up	3x	5x	20x	16x
1%	Scanning	48.0s	348.7s	3301.3s	19756s
	Indexing	7.8s	28.1s	41.0s	99.1s
	Speed-up	6x	12x	81x	199x

- Speed-up with indexing: 3x – 199x

15

But challenges remain…

- Index construction time
  - 3 min/3.6GB
  - 23 min/27GB
  - 3 hr/208GB
  - > 12hr/1.7TB

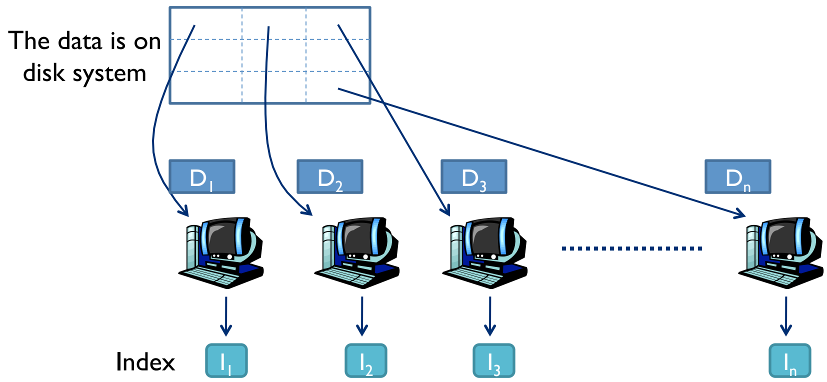


❑Solution:

– Building indexes in parallel!

16

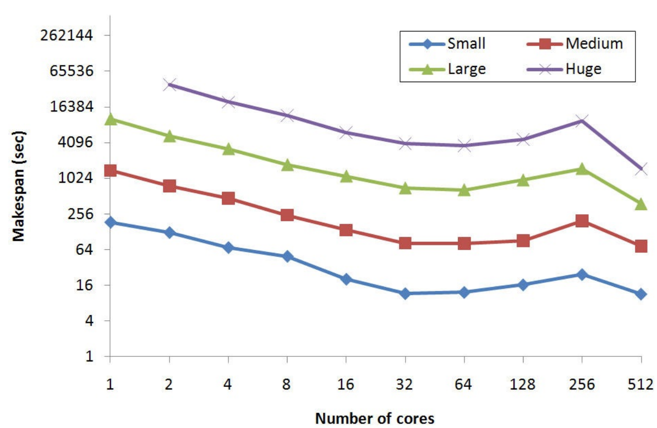
Parallel Index Construction



- Split and assign data blocks to multiple processors

17

Performance with Parallelism



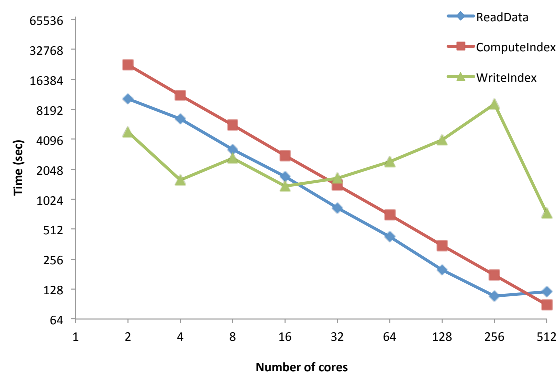
Hopper2 @ LBNL

- **~6000 nodes**
- **24 cores**
- **32-64 GB memory**
- **Lustre file system**

- Parallelism improves performance, but
- Why the benefit disappears after a certain parallelism factor?

18

Index Construction Time Breakdown

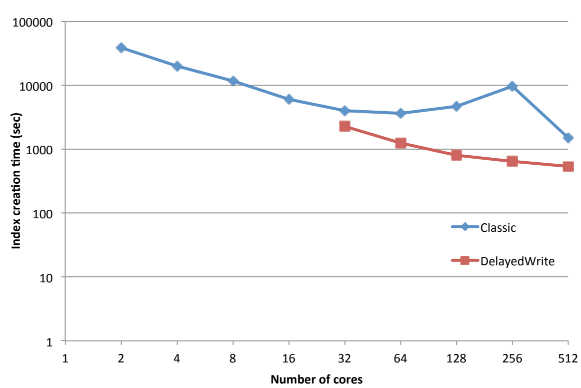


- Write performance shows little improvement!
- Why? Collective writes → Sync overhead

19

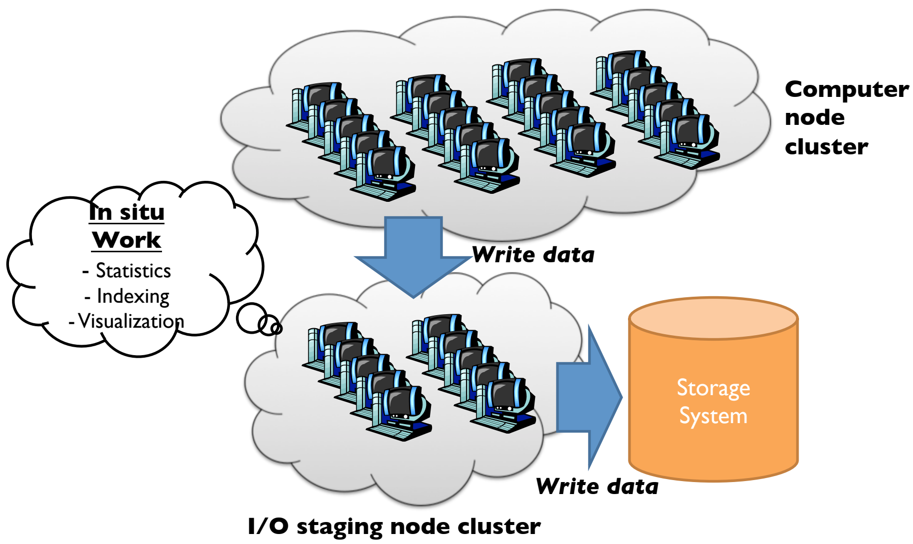
Optimization: Delayed Writes

- Reduce number of synchronizations!
  - Delaying writing index whenever possible
  - Retain created indexes in memory, then write them together



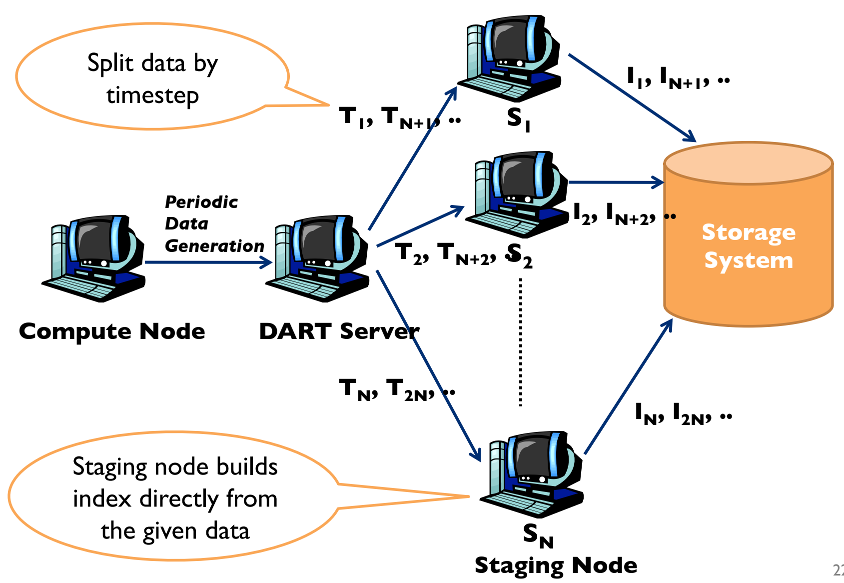
20

Cluster with Dedicated Staging Nodes



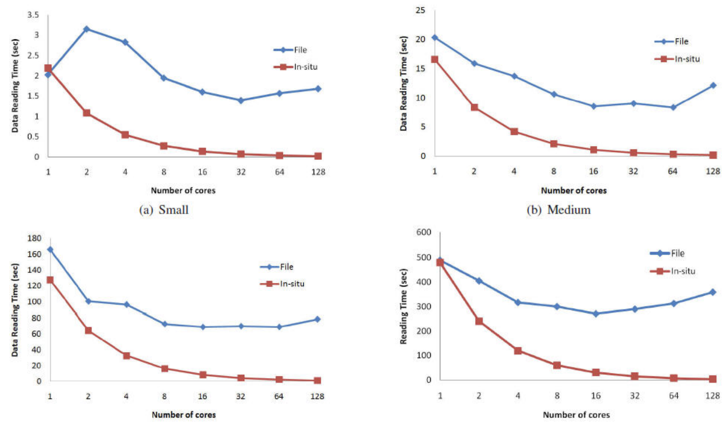
21

Experiments for In Situ Indexing



22

Reading Time



(a) Small (b) Medium (c) Large (d) Large2

Franklin @ LBNL

- ~10000 nodes
- 8 cores
- 8 GB memory
- Lustre file system

Small: 3.6GB
Medium: 27GB
Large: 208GB
Large2: 173GB

- Getting data from another processor (in situ) is faster than getting data from disk

23

Summary

- Indexing dramatically reduces query time
  - But expensive with 12+ hours for 1 TB data
- Parallelism offers performance improvement for building index
  - But collective writes causes random delay
  - Delayed write optimization can mitigate the delay
- In situ indexing improves performance by significantly reducing data read time

24

Lessons Learned

- Avoiding synchronization
  – One delayed processor causes severe delay in writing
  – It is fine to delay writing index blocks if the base data is safely stored already
- Choosing a moderate number of processors
  – Performance benefits are not linear!
  – Finding sweet spot may be interesting (maybe GLEAN could help)
- Tuning file system parameters
  – For example, striping count has direct performance impact to some extent

25

John Wu John.Wu@nersc.gov
FastBit http://sdm.lbl.gov/fastbit/
FastQuery http://portal.nersc.gov/svn/fq/
ADIOS http://www.olcf.ornl.gov/center-projects/adios/

QUESTIONS?

26