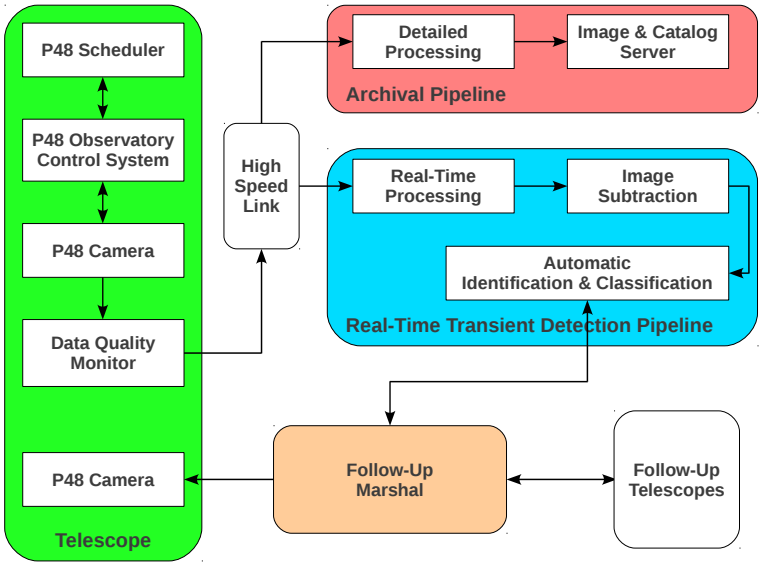
Distributed Caching for Processing Raw Arrays

Weijie Zhao[1], **Florin Rusu**[1,2], Bin Dong[2], Kesheng Wu[2], Anna Y. Q. Ho[3], and Peter Nugent[2]

[1]University of California Merced, [2]Lawrence Berkeley National Laboratory, [3]CalTech

July 2018

Palomar Transient Factory (PTF)

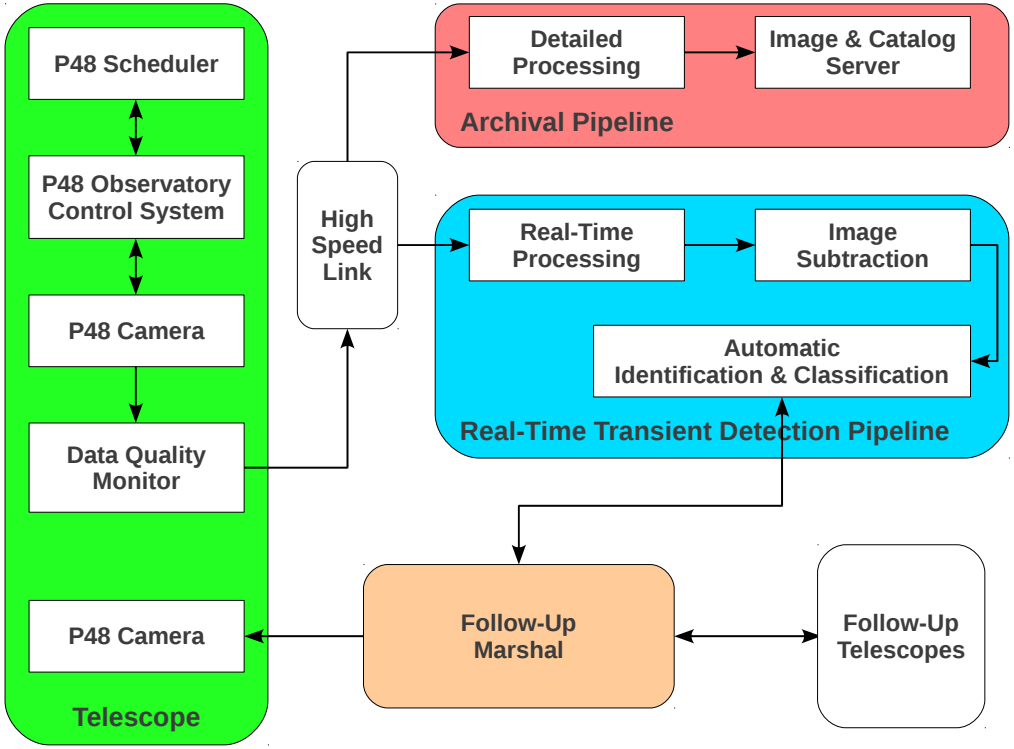


- 2,000-4,000 images 2,048x4,096 pixels per night
- 60-100 GB per night
- Complete processing of an image set within 30-45 minutes of aquisition

Automatic Transient Identification & Classification

- 1-1.5 million candidates extracted per night; 10,000 every 45 minutes; 30-150 are real
- Real-or-bogus classification: Is the candidate real?
- What is the transient type of a real candidate: VarStar? SN/Nova? circumnuclear event? asteroid?

Array Construction



- Image $\xrightarrow{\textit{Image processing}}$ Objects $\rightarrow$ 3-D array [`ra, dec, time`]
- Array is chunked and stored in a shared-nothing array database
- Array is sparse and skewed

Outline

Raw PTF Data (FITS, HDF5, CSV)

A: Standard FITS Format

Figure FITS-1

B: Binary Table Extension

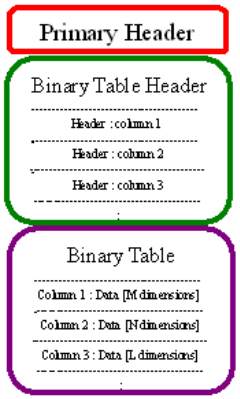


Figure FITS-2

```
objID,skyVersion,run,rerun,camcol,field,obj,mode,nChild,type,clean,probPSF,insideMask,fl
1237645942366274027,2,109,301,3,114,491,1,0,3,1,0,0,217164284160,226.89418,0.13062,1969.
1237645942366274688,2,109,301,3,114,1152,1,0,3,0,0,0,281543964623104,145.553284,0.520364
1237645942366274694,2,109,301,3,114,1158,1,0,6,0,1,0,281543964623616,225.317307,0.507917
1237645942370140312,2,109,301,3,173,152,1,0,6,1,1,0,158398662443776,720.658875,4.751435E
1237645942370140314,2,109,301,3,173,154,1,0,6,1,1,0,68988047872,734.537415,4.991623E-3,1
1237645942370140321,2,109,301,3,173,161,1,0,6,1,1,0,193585182019600,771.662109,3.504038E
1237645942370140322,2,109,301,3,173,162,1,0,3,0,0,0,144572653934612752,763.089355,0.5900
1237645942370140323,2,109,301,3,173,163,1,0,3,0,0,0,144431916446257424,797.722046,0.5742
1237645942370140324,2,109,301,3,173,164,1,0,3,0,0,0,316728370401624,759.693542,0.430969,
1237645942370140325,2,109,301,3,173,165,1,0,3,0,0,0,144431916446257424,772.056763,0.7608
1237645942370140327,2,109,301,3,173,167,1,0,6,1,1,0,68987912704,813.796387,0.016672,693.
1237645942370140330,2,109,301,3,173,170,1,0,6,1,1,0,35253360136208,845.047607,5.510448E-
1237645942370140331,2,109,301,3,173,171,1,0,3,0,0,0,105622104310104,840.993713,0.351185,
1237645942370140332,2,109,301,3,173,172,1,0,6,0,1,0,387097114444048,841.738831,0.33631,1
1237645942370140503,2,109,301,3,173,343,1,0,6,1,1,0,68987912960,975.684814,0.068882,665.
1237645942370140504,2,109,301,3,173,344,1,0,3,1,0,0,217164284160,984.625671,0.367924,97.
1237645942370140505,2,109,301,3,173,345,1,0,6,1,1,0,68987912448,1008.3241,0.055251,295.5
1237645942370140507,2,109,301,3,173,347,1,0,3,1,0,0,68987912448,1098.70068,0.191982,131.
1237645942370140509,2,109,301,3,173,349,1,0,6,1,1,0,72092847397929232,1136.47461,0.04868
1237645942370140510,2,109,301,3,173,350,1,0,3,1,0,0,35184640524816,1147.8905,0.054839,36
1237645942370140512,2,109,301,3,173,352,1,0,3,1,0,2,68988043520,1212.91077,0.191284,584.
1237645942370140514,2,109,301,3,173,354,1,0,6,1,1,0,68987912192,1220.56543,0.065685,318.
1237645942370140516,2,109,301,3,173,356,1,0,3,1,0,0,2278815830856,1231.65027,0.151327,78
1237645942370140518,2,109,301,3,173,358,1,0,3,1,0,0,35255507620624,1255.73218,0.109877,3
1237645942370140519,2,109,301,3,173,359,1,0,6,0,1,0,545426755424528,1270.13306,0.565565,
```

- A file for each night; file ranges overlap
- High object cardinality variance across files
- Files stored across a distributed cluster
- Load and partition inside array database before query

Contributions

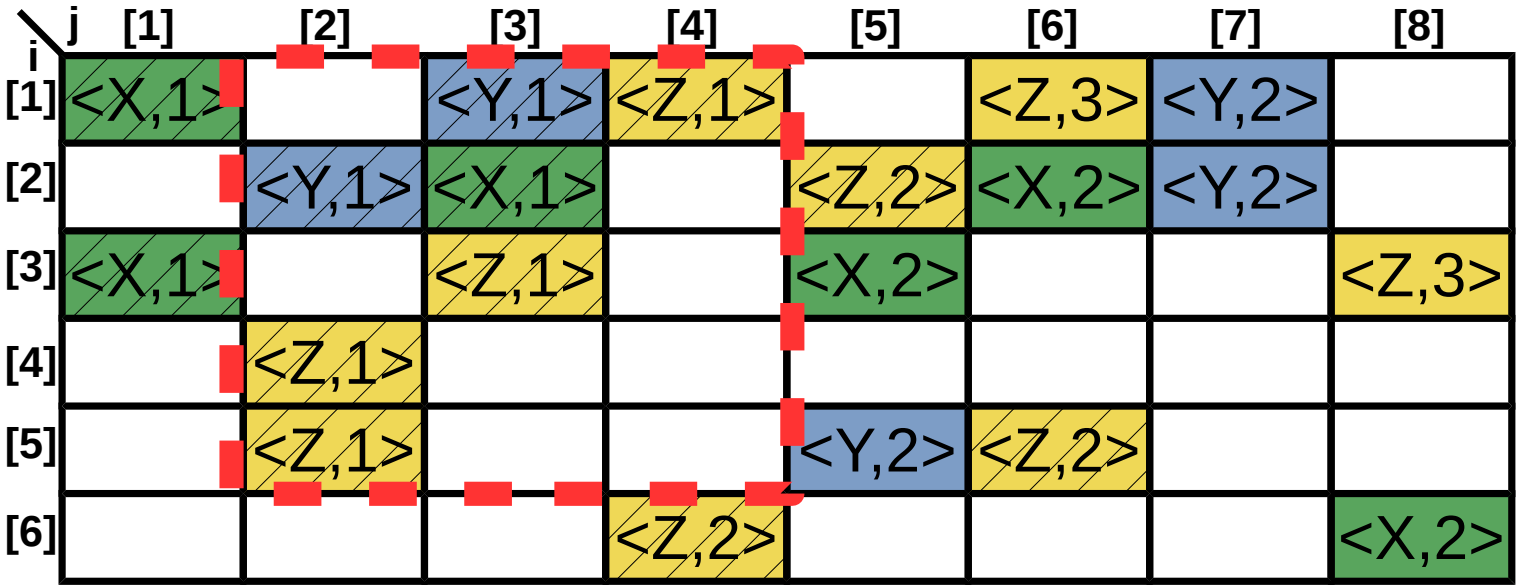
- Design query-driven distributed caching framework for raw arrays
  - Identify cells to be cached locally and incrementally from each input file
  - Assign cells to nodes such that dependent cells are collocated
- Design an evolving R-tree index that refines the chunking of a sparse array to efficiently find the cells contained in a given subarray query
- Propose efficient cost-based algorithms for distributed cache eviction and placement that consider a historical query workload
- Evaluate experimentally distributed caching over raw arrays on two real sparse arrays with more than 1 billion cells stored in three file formats—FITS, HDF5, and CSV

Related Work

- Raw or in-situ data processing over CSV and JSON files
  - NoDB, SCANRAW, OLA-RAW, VIDa, SLALOM, Proteus
- Raw array processing in SciDB (HDF5)
  - ArrayBridge
- Caching of raw semi-structured data (JSON, Parquet)
  - NoDB, ReCache, vertical partitioning, invisible loading
- Distributed caching in relational databases and web systems
  - LRU-K, DBMIN, LRFU, LRU-Threshold, Lowest-Latency-First, and Greedy-Dual-Size
- Distributed caching in Hadoop and Spark

Outline

Raw Array Data Model

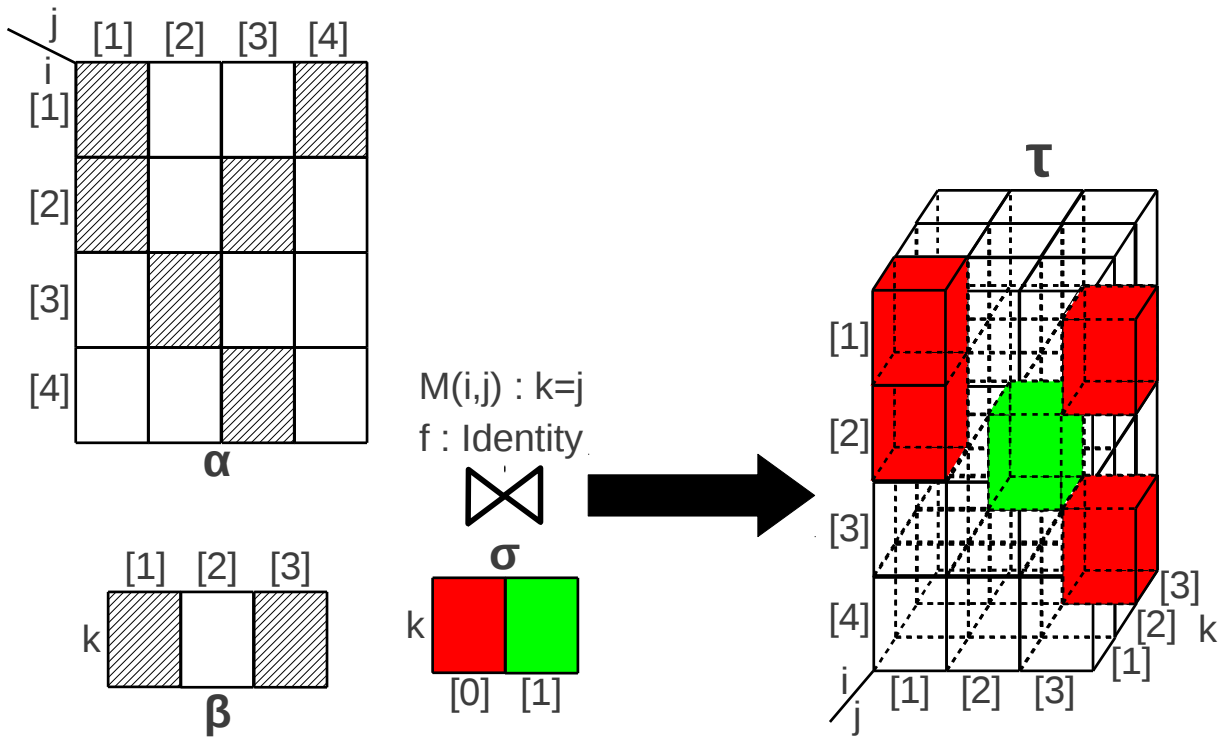


- Raw array `A<n:char,f:int>[i=1,6;j=1,8]` consisting of 7 files distributed over 3 nodes
- Attributes `n` and `f` correspond to the node and the file id on the node, e.g., `<X,2>` is a cell in the second file on node X
- The cell color also shows the server on which the file is stored
- The dashed rectangle specifies a query

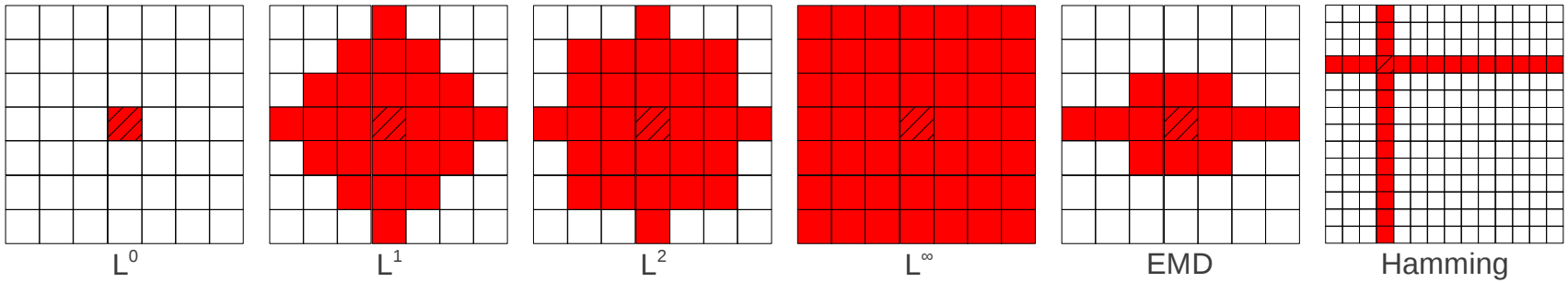
Array Similarity Join

- SELECT f INTO τ FROM α SIMILARITY JOIN β ON $\mathcal{M}$ WITH SHAPE σ

f : computation function, τ : result array, α, β : input arrays, $\mathcal{M}$: array mapping function, σ : similarity shape array

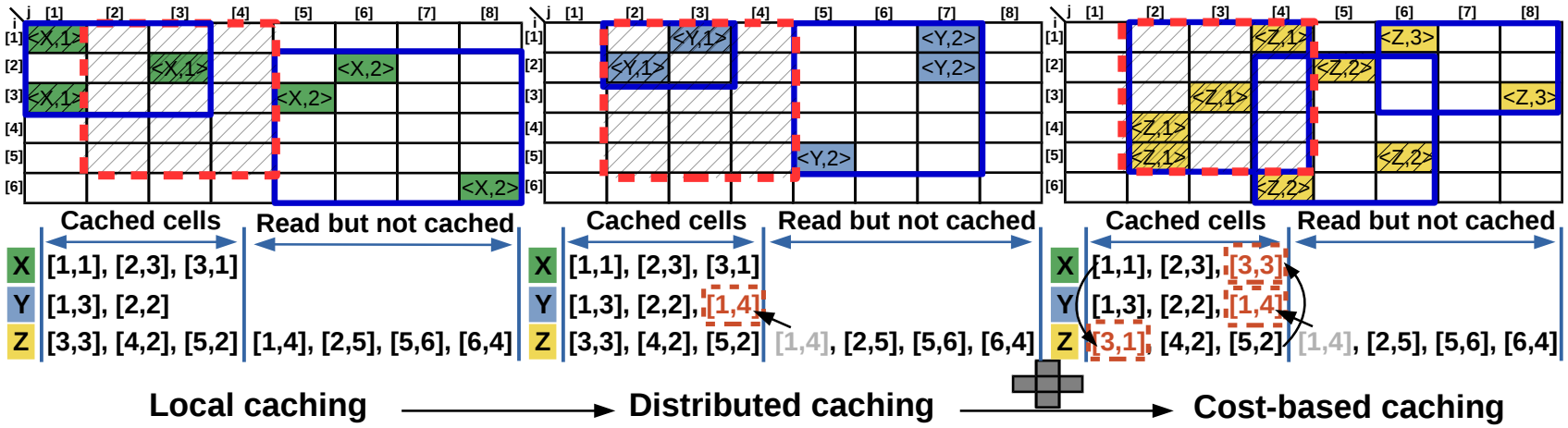


Similarity Shape Array for Popular Distance Metrics



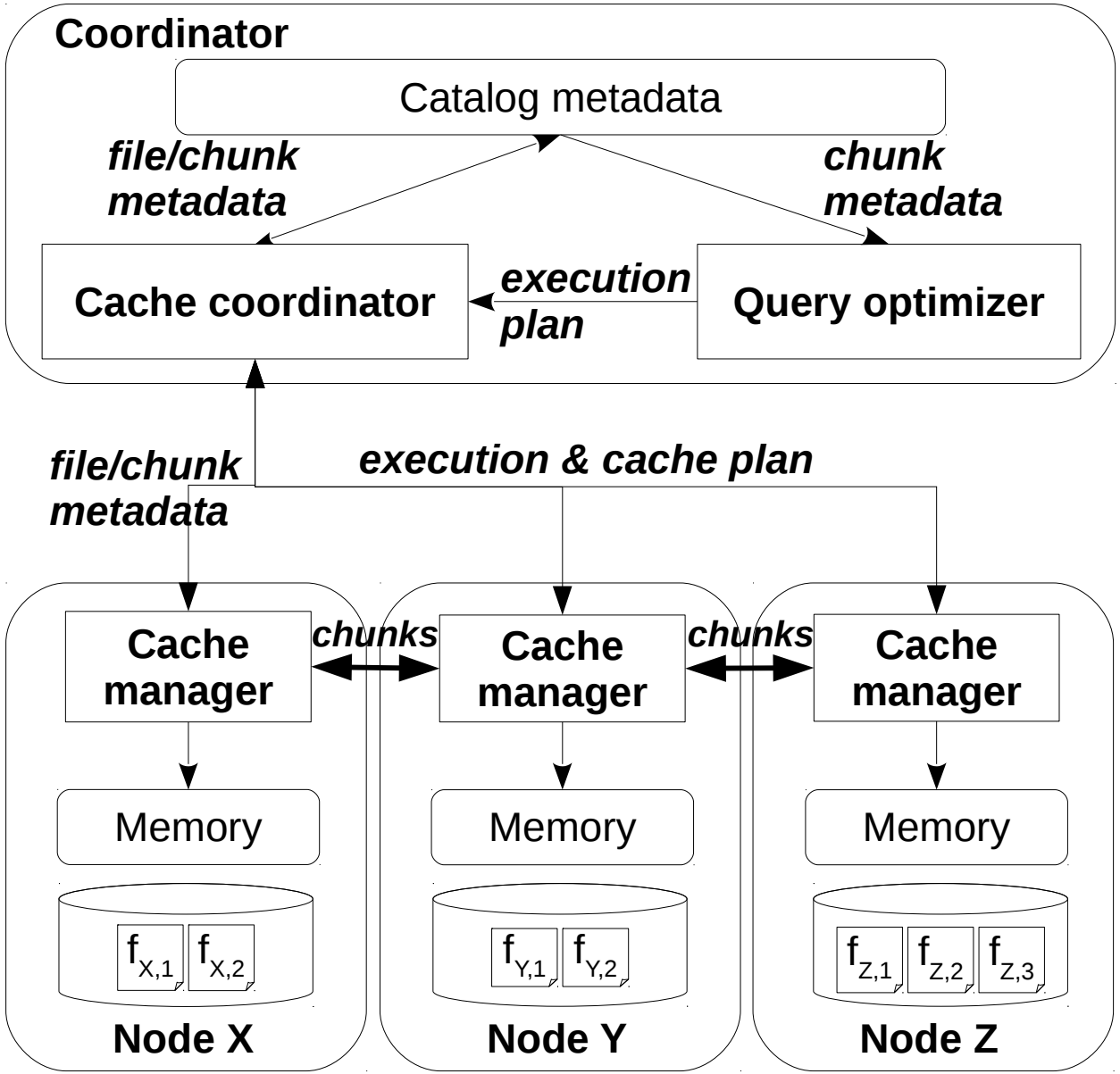
Outline

Raw Array Distributed Caching Approach



- Build in-memory chunks incrementally based on the query (workload-driven chunking)
- Create higher granularity chunks for the entire file
- Design a global cost-based caching mechanism
- Combine query processing with cache replacement

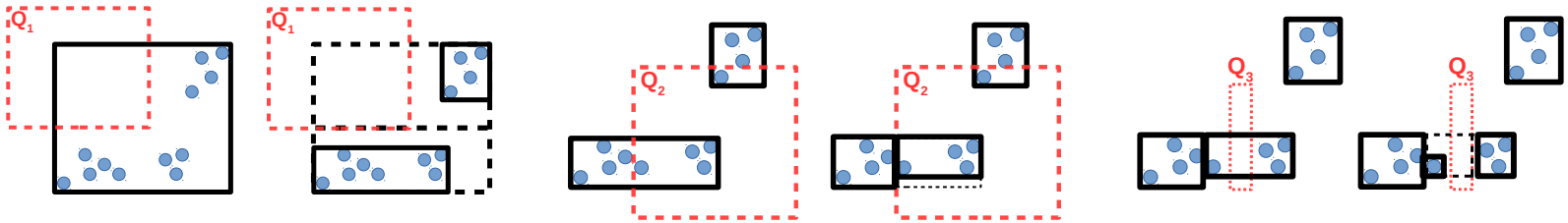
Distributed Caching Architecture



Raw Array Chunking

- Original raw array and resulting chunked array

- Query-driven chunking



Chunk Split Algorithm

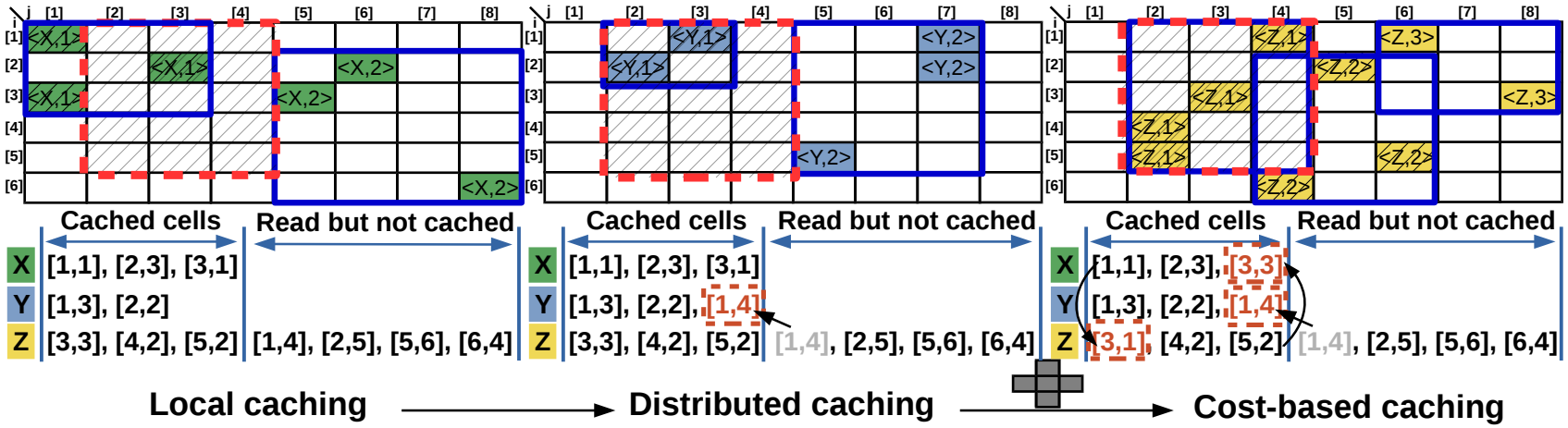
Input: Chunk α with bounding box $\mathcal{BB}_\alpha$ that intersects query subarray Q; Minimum number of cells threshold *MinC*

Output: Chunks β and γ after splitting α

1: **if**(cells in $\alpha < MinC$) **and** ($\exists$ cell in $\alpha \in Q$) **then return**
2: min_vol $= +\infty$
3: **for each** boundary $b \in Q$ that intersects with $\mathcal{BB}_\alpha$ **do**
4: $(\beta_b, \gamma_b) \leftarrow$ split cells in α into two sets by boundary b
5: **if** vol(β_b) + vol(γ_b) $<$ min_vol **then**
6: min_vol $\leftarrow$ vol(β_b) + vol(γ_b)
7: $\beta \leftarrow$ bounding_box(β_b), $\gamma \leftarrow$ bounding_box(γ_b)
8: **end if**
9: **end for**

- Build an evolving R-tree incrementally based on queries
- When to split? **Cell number threshold or no cells at all**
- How to split? **Always split into two chunks**

Cost-Based Caching



- Find optimal caching plan
  - Centralized setting: minimize cache misses
  - Distributed setting: minimize network misses
- Weigh chunks based on how they are accessed in the query workload
  - Access frequency: cache eviction
  - Co-locality: cache placement

Cache Eviction

- Scan a file entirely even if one accessed chunk is not cached
- $cost_{evict}(Q_l, f_i, \{C_j\}) = w_{Q_l} \cdot \frac{size(f_i)}{\sum size(uncached\ C_j)}$

Input: Cache state as set of triples $S = \{(Q_l, f_i, \{C_j\})\}$ consisting of the chunks j accessed from file i at query l; Set of pairs $(f_i, \{C_j\})$ consisting of the chunks j accessed from file i at the current query Q_{l+1}; Cumulated cache budget $B = \sum_k B_k$ consisting of local budgets B_k at each node

Output: Updated cache state $S' = \{(Q_{l+1}, f_i, \{C_j\})\}$

1: $S' \leftarrow \{(Q_{l+1}, f_i, \{C_j\})\}$
2: **while** $budget(S') < B$ **do**
3: Extract triple $t = (Q_l, f_i, \{C_j\})$ from S such that $budget(t \cup S') \leq B$ and $cost(t)$ is maximum
4: $S' \leftarrow S' \cup t$
5: $S \leftarrow S \setminus t$
6: Increase cost of triples $t' \in S$ that contain chunks in t
7: **end while**

Cache Placement

- Piggyback on the replication induced by query execution
- Preserve a single copy of every chunk
- Maximize co-locality of correlated chunks
- $cost_{place}(C_i, n, P', W) = \sum_{Q \in W} w_Q \cdot |C_j \in P'_n \wedge (C_i, C_j) \in Q|$

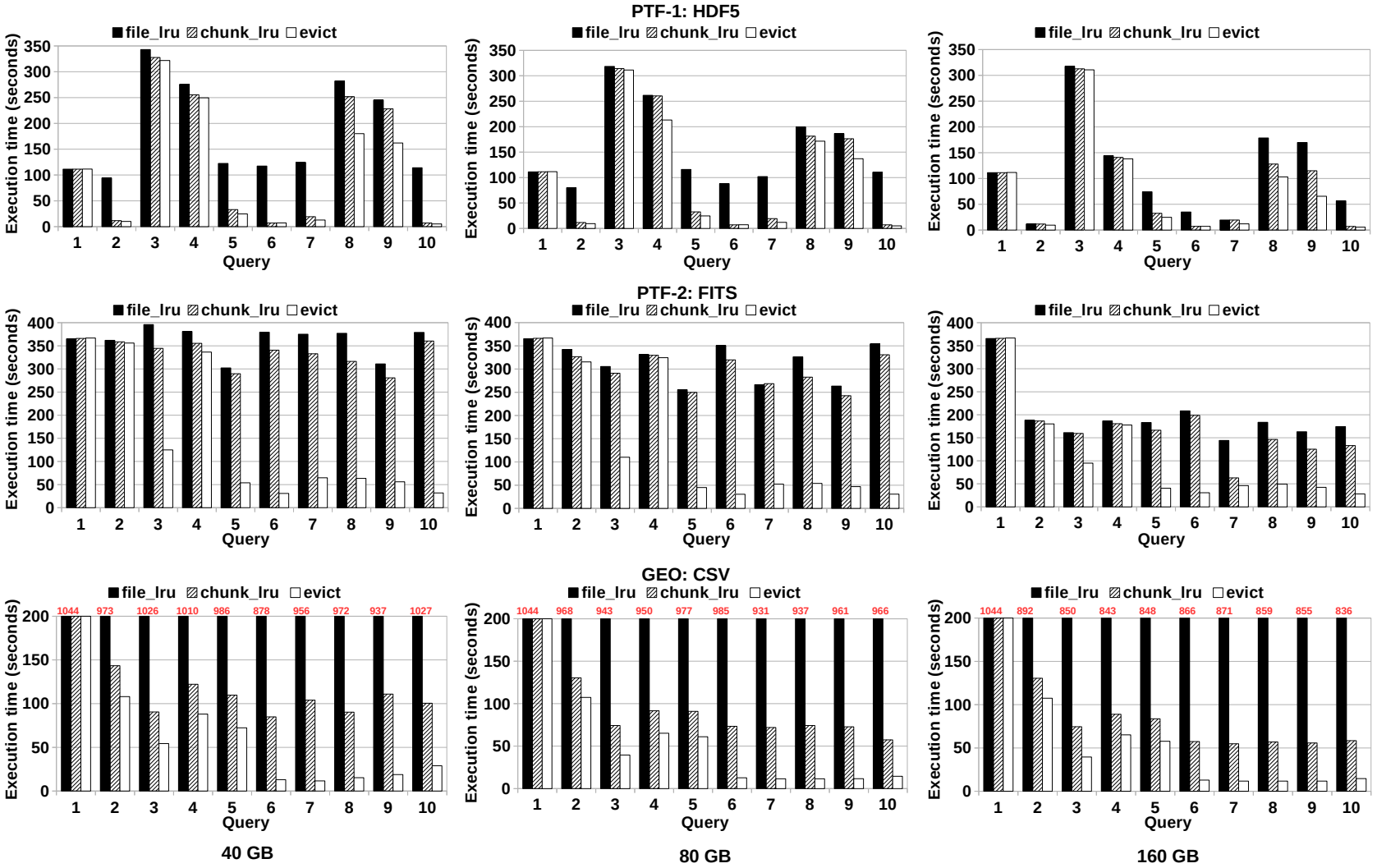
Input: Set $W = \{(Q_l, \{(C_i, C_j)\})\}$ consisting of chunk pairs (i, j) that join at query l; Set of locations $P = \{(C_i, \{N_k\})\}$ specifying all the nodes k that have a copy of cached chunk i at current query Q_{l+1}; Cache budget B_k at node k

Output: Updated locations $P' = \{(C_i, N_k)\}$

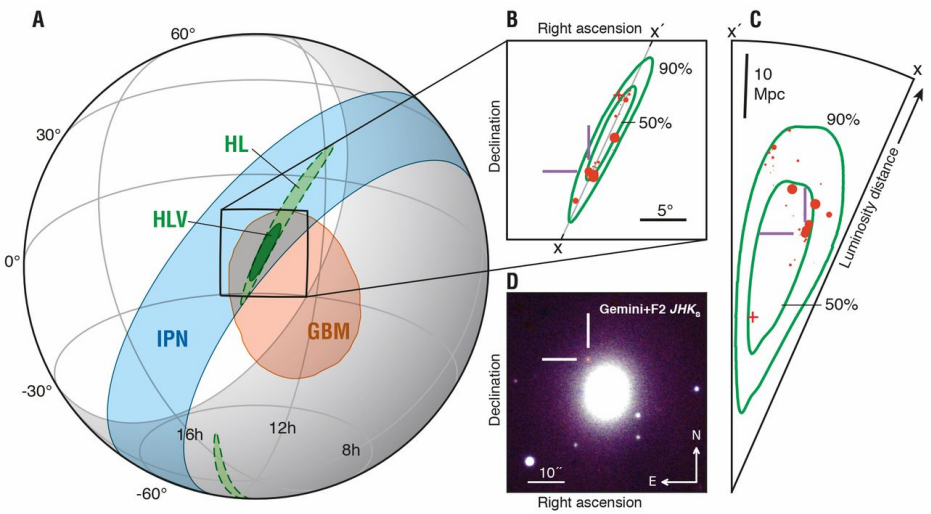
1: $P' \leftarrow \{p \in P\}$, where p has no replicas
2: **for each** $p = (C_i, \{N_k\}) \in P$ with multiple replicas **do**
3: Select node $n \in \{N_k\}$ such that $budget_n(C_i) \leq B_n$ and $cost(C_i, n, P', W)$ is maximum
4: $P' \leftarrow P' \cup (C_i, n)$
5: **end for**

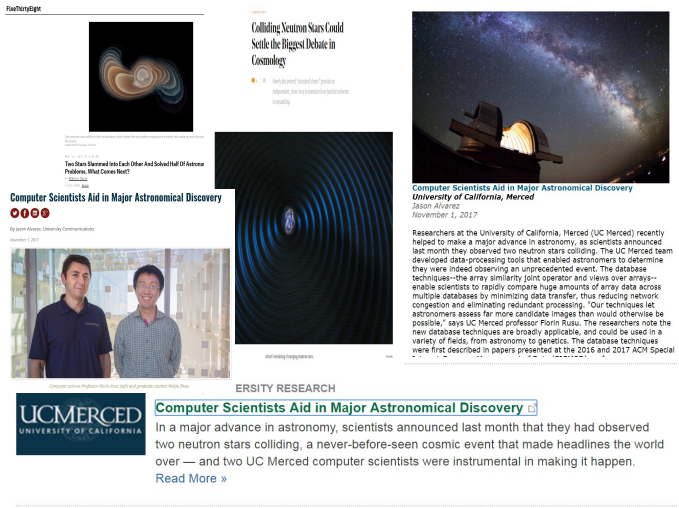
Outline

Experiments: Query Execution Time



First-Ever Neutron Star Merger Observation





- Produces gravitational waves and turns out to be the origin of heavy elements, including gold
- **Science** article: *Illuminating gravitational waves: A concordant picture of photons from a neutron star merger*

Conclusions

- Design query-driven distributed caching framework for raw arrays
- Design an evolving R-tree index that refines the chunking of a sparse array to efficiently find the cells contained in a given subarray query
- Propose efficient cost-based algorithms for distributed cache eviction and placement that consider a historical query workload

Thank you!

Questions?