# Inverted Indices for Particle Tracking in Petascale Cosmological Simulations

Daniel Crankshaw
The Johns Hopkins University
Dept. of Computer Science
Baltimore, MD 21218
+1-650-269-0846
dcrankshaw@jhu.edu

Bridget Falck
Institute of Cosmology and Gravitation
University of Portsmouth
Portsmouth, PO1 3FX, UK
+44 (0)23 9284 3137
bridget.falck@port.ac.uk

Alexander S. Szalay
The Johns Hopkins University
Dept of Physics and Astronomy
Baltimore, MD 21218
+1-410-516-7217
szalay@jhu.edu

Randal Burns
The Johns Hopkins University
Dept. of Computer Science
Baltimore, MD 21218
+1-410-516-7708
randal@cs.jhu.edu

Tamás Budavári
The Johns Hopkins University
Dept of Physics and Astronomy
Baltimore, MD 21218
+1-410-516-0643
budavari@pha.jhu.edu

Jie Wang
Nat. Astron. Obs. Of China
Datun Road, ChaoYang
Beijing, China
+86-010-64888708
jie.wang@nao.cas.cn

## ABSTRACT

We describe the challenges arising from tracking dark matter particles in state of the art cosmological simulations. We are in the process of running the Indra suite of simulations, with an aggregate count of more than 35 trillion particles and 1.1PB of total raw data volume. However, it is not enough just to store the particle positions and velocities in an efficient manner – analyses also need to be able to track individual particles efficiently through the temporal history of the simulation. The required inverted indices can easily have raw sizes comparable to the original simulation.

We explore various strategies on how to create an efficient index for such data, using additional insight from the physical properties of the particle motions for a greatly compressed data representation. The basic particle data are stored in a relational database in course-grained containers corresponding to leaves of a fixed depth oct-tree labeled by their Peano-Hilbert index. Within each container the individual objects are sorted by their Lagrangian identifier. Thus each particle has a multi-level address: the PH key of the container and the index of the particle within the sorted array (the slot).

Given the nature of the cosmological simulations and choice of the PH-box sizes, in consecutive snapshots particles can only cross into spatially adjacent boxes. Also, the slot number of a particle in adjacent snapshots is adjusted up or down by typically a small number. As a result, a special version of delta encoding over the multi-tier address already results in a dramatic reduction of data that needs to be stored. We follow next with an efficient bit-compression, adapting to the statistical properties of the two-part addresses, achieving a final compression ratio better than a factor of 9. The final size of the full inverted index is projected to be 22.5 TB for a petabyte ensemble of simulations.

## Categories and Subject Descriptors
H.2.8 [**Database Applications**]: Scientific Databases, Indexing methods, Spatial Indexing

## General Terms
Algorithms, Design.

## Keywords
Cosmological N-body simulations, Inverted Index

## 1. INTRODUCTION

The next generation of astronomical surveys will measure millions of galaxy positions and probe an ever-increasing volume of the sky. We now know the value of cosmological parameters within an accuracy of a few percent, at which level subtle effects become dominant and parameter estimation becomes increasingly sophisticated. In order to understand structure formation at the largest scales, data-intensive simulations are needed to keep up with data-intensive observations.

Indra is a suite of large-volume cosmological N-body simulations that will provide excellent statistics of the large scale features in the distribution of dark matter while at the same time resolving the nonlinear evolution of structure. In total, the Indra simulations will save over a Petabyte of data, which will be primarily in the form of the positions and velocities of dark matter particles at many time steps as the simulation evolves. To connect these dark matter particles to observations of galaxies, we will identify and store the dark matter halos – the nonlinear collapsed structures in which galaxies reside – and the lists of particles that define each halo. The challenge is that access patterns require more than one type of indexing: one is based upon the spatial location of the particles, and the other aggregates the whole time history for each particle, indexed by their unique ID number, an inverted index for 35 Trillion particles! This paper presents an efficient solution to storing and indexing these particles and their time histories for cosmological simulations.

## 1.1 Simulations and Cosmic Variance

Large cosmological simulations provide the bridge between linear theory for the evolution of small fluctuations in the Universe and today's complex structure as seen in large-scale surveys. In order to determine what the Universe was like at early times, astrophysicists have been using large numerical simulations based on first principles and a set of initial cosmological parameters to predict how the Universe would look today. By computing similar statistics on the observations and on the simulations, we can "invert" today's observable Universe and get a better understanding of the precise initial conditions. But observations of the Universe have a major problem: there is only one Universe, and we cannot change our point of reference and sample it from different locations. This effect, "cosmic variance," poses an ultimate limit on the uncertainties we can derive purely from observations. On the other hand, we can use an ensemble of cosmological simulations to predict the uncertainties in the cosmological parameters and their covariances.

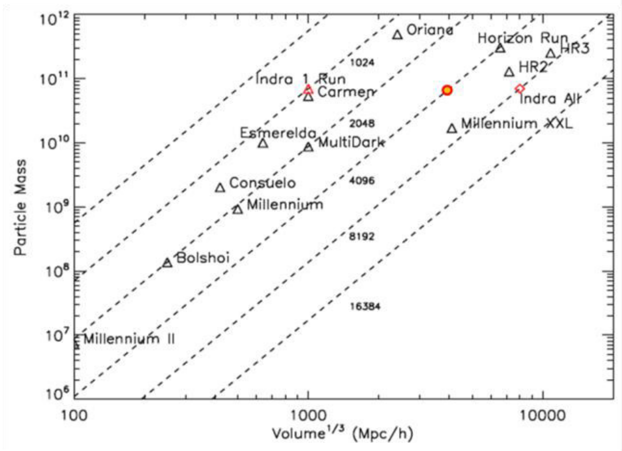

**Figure 1.** A summary of the current state of the art in large-scale cosmological simulations. The symbols in red denote Indra: the triangle is one simulation, the yellow circle is the current state of 80 runs completed, and the diamond is the whole 512 run suite. The horizontal axis is the linear size of the total box, and the vertical scale shows the mass resolution.

The spatial power spectrum of cosmological fluctuations contains a feature, an acoustic resonance frequency, corresponding to the size of the Universe when it was 300,000 years old. This feature is quite large, at a spatial scale of 128 Mpc/h [5]. Here, h is a standard notation for the uncertainty in Hubble's constant, at the core of all distances in cosmology. h is the precise value, measured in units of 100 km/s/Mpc. 1 Megaparsec is a convenient unit of distance for cosmology; it is $3.08 \times 10^{24}$ cm. One of the most important cosmological questions today is the determination of the precise details of the resonant peak in the spectrum, and how this is correlated with different cosmological properties. As the spatial scale of this feature is quite close to the box sizes in many of the current simulations (see Figure 1), in any single simulation there are very few independent modes on this scale and the effective degrees of freedom is very small. As a result, any measurements on these scales are quite uncertain by necessity of design.

Today's simulations are pushing toward one of two competing drivers: bigger box sizes or higher resolution (Figure 1)**.** Both require excessive supercomputer resources (hundreds of thousands of cores and Terabytes of memory) and present significant data challenges – so they are saving only a few snapshots and running a single realization. While this is certainly useful for certain numerical experiments, it does not solve the cosmic variance problem. Even chopping the largest simulations into smaller sub-volumes and using these to approximate an ensemble average on smaller scales does not quite do the job.

Due to the non-linear nature of the simulations and the resulting mode-mode coupling, the different subvolumes of a single large simulation are still correlated. Any ensemble averaging over these correlated subvolumes is a difficult task, and the estimation of covariances is impacted by the unknown higher order correlations. These averages will converge very weakly to the true ensemble average, in the limit of infinitely large box size.

The most efficient way to lower the statistical uncertainties in the covariances is to run many independent simulations on the scale of the sub-volumes, each with different initial random numbers. With this approach, we will get a much faster convergence and true independence. There is a clear niche for an ensemble of simulations which individually do not strive to be the largest, but in their aggregate scale they will be superior to any other simulation available today. This is the main motivation for our project, the Indra suite of simulations.

## 1.2 The Indra Suite of Simulations

Indra consists of a suite of 512 N-body simulations, each with a little over 1 billion ($1024^3$) dark matter particles in a 1 Gpc/h periodic box. Initially, we have one particle in every 1 Mpc/h subvolume. The initial conditions (the random amplitudes and phases for each Fourier mode) in each simulation are based upon the same cosmological parameters, but with different random seeds, providing an excellent statistical characterization of the very large scale modes of the matter density field and with a mass resolution of about $10^{11}$ solar masses per particle.

The simulations are using the Gadget [14] code, running efficiently over multiple parallel nodes using MPI. The simulations start at very early times and finish at an epoch corresponding to the current age of the Universe. During each run 64 snapshots are generated, containing particle IDs, positions and velocities. Besides the snapshots we also output the complex Fourier amplitudes for the innermost core of the Fourier transform of the density, representing all spatial scales larger than 10 Mpc/h. Furthermore, in each snapshot we run a halo-finder, using a friends-of-friends algorithm [2] to identify groups of dark matter particles ("halos") which will be the sites of galaxy formation.

A single Indra simulation snapshot contains 1.07B particles. With 64 snapshots, in a single simulation we need to store the positions, velocities and IDs of 68.5B particles. By the time we reach the whole ensemble of 512 independent realizations, we will have 35 trillion separate particle instances to deal with in the data set.

Positions and velocities are output as single precision floating point numbers, and the particle ID is a 4 byte integer. The raw data volume is 2.24TB per simulation, 1.15PB for the whole ensemble. To date, we have completed 80 runs, resulting in 5.5 trillion particles and 180TB of raw data.

The volume and resolution of the Indra simulations will allow studies of structure formation from the largest scales all the way into the nonlinear (small scale) regime. We will additionally be able to run a few high-resolution re-simulations [7] to capture more precisely the galaxy-scale formation of structure. For the purposes of this paper we will use a down-sampled version of a single realization of Indra, where the number of particles is only $512^3$.

## 1.3 From Simulations to Cosmological Laboratories

Traditionally, numerical simulations were distributed by enabling people to download the binary snapshot files, which worked fine as long as the files were a few GB per snapshot and there were not too many of them. However, the full suite of Indra simulations will produce a bit over a Petabyte of data. Few universities and national laboratories have the resources and expertise to manage and analyze such a large dataset, and moving hundreds of Terabytes over the network is not currently feasible. With that in mind, we are setting up Indra as a public numerical laboratory, where the data is optimally organized for the necessary analysis patterns and the computations performed where the data is located. This will be part of the Data-Scope instrument at the Johns Hopkins University, which has been designed specifically for data-intensive analyses.

The analysis of cosmological simulations often involves accessing relatively small regions of space out of a very large box or filtering the data according to certain properties. Databases, with their excellent indexing properties, turned out to be surprisingly well-suited to this purpose. The database built on top of the Millennium simulation [8] quickly became the world's most popular reference simulation resource for cosmology; more than a thousand people are running database queries over the search engine regularly. A clever data structure inside the database enables extremely fast searches and aggregations over the merger trees of galaxies, and a user-defined function builds dynamic light cones out of the simulation on-the-fly.

However, the Millennium simulation database does not store the full snapshots of particle positions and velocities but instead contains primarily halos, which are collapsed regions identified in the simulation using the dark matter particle positions. Another simulation database, MultiDark [12], was able to store only a few snapshots of particle data because of the huge storage volume required; additionally, queries on its particle table are admittedly time-consuming, limiting its usefulness.

With Indra, we will be storing all the particle positions and velocities, for all 64 snapshots and all 512 simulation runs, totaling 35 trillion particles. There will also be halo catalogs, linked to the particle data, and the complex Fourier modes of the density field, which are most naturally stored as a cubic grid. The database for the Indra suite of simulations will require efficient storage of the particle positions and velocities, and cosmological analyses will require the ability to track individual particles through the temporal history of the simulation as they collapse into halos. New data structures and indexing schemes are needed for a Petabyte of cosmological simulation data.

This paper proceeds as follows. In Section 2, we describe in some detail the storage scheme designed for the Indra simulations and the requirements for the efficient analysis of the data, motivating the use of inverted indices as well as a compact storage model for the base date itself. We describe the indexing scheme in Section 3, showing how it greatly speeds up some typical queries and can be built on-the-fly. In Section 4 we explain how we exploit the dynamics of the simulation to compress the indices, which could easily have raw sizes comparable to the original simulation. Finally in Section 5 we conclude with discussion of ways to improve the compression and compute indices as we dynamically load the simulation one snapshot at a time.

## 2. THE INDRA STORAGE MODEL

The data for each individual Indra simulation will be stored in a relational database. At JHU we have been using a cluster of very high performance database servers, all running Microsoft SQL Server 2008. SQL Server offers an extremely flexible mechanism to augment its properties with user defined functions and data types, capable of implementing very complex objects with their properties and methods, all accessible through SQL. The database engine provides automatic parallelism (as long as the data is laid out well), and efficient query plans. Over the years we have developed an elaborate spatial indexing library [9] that enables the use of space filling curves to organize and search data, and maps onto range queries over the high performance B-trees inside the database engine.

The data is organized into three main tables: the Particle table contains the particle ID, position, and velocity of every particle for all 64 snapshots, labeled by snapnum, in the range of [0,63]; the Halo table contains the halo ID, the list of particle IDs that comprise each halo, and various derived halo properties, for all halos at all snapnums; and the Fourier table contains the Fourier modes of the large-scale density field saved at a high temporal resolution. The full suite of Indra simulations will thus consist of 512 separate databases, each with its own Particle, Halo, and Fourier tables, plus some small tables of metadata containing the physical units of the simulation, the correspondence between snapshot number and cosmological epoch, etc. These databases can (and will) be stored on separate database servers, distributed over a large cluster with a fast interconnect.

Each of the main data tables will make use of a user-defined data type called SqlArray [4], which allows blobs of binary data ordered in multi-dimensional arrays to be stored as one item in a row. For the particle data, this greatly reduces the number of required rows, considering that a schema with one row per particle results in over 68 billion rows for just one of the 512 Indra simulation runs. For the halo data, we will store the IDs of the particles making up the halo also as a SqlArray, in the same row as the rest of the halo properties. This halo membership array can then be used to link to the particle data as well as calculate halo properties on the fly. The array-based storage is also an optimal way to store the Fourier modes of the density field and integrates well with common math libraries for the calculation of Fast Fourier Transforms directly in the database. The SqlArray data-type is also used by several other projects, namely the turbulence databases at JHU, currently containing more than 100TB of data [11].

In analyzing cosmological simulations, one is often interested in a particular region of the simulation cube – i.e., halos or collapsed structures, voids or low-density regions, particles near halos, light cones, etc. Many common analyses require identifying particles or halos within localized regions of the simulation volume. Thus instead of randomly assigning particles to different arrays, we group the particles into "buckets" organized according to a space-

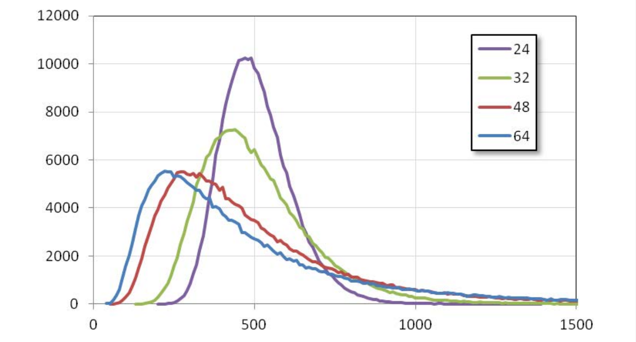

**Figure 2.** The distribution of the number of particles per bucket as a function of time (labeled with snapnum). The average number of particles is 512 per bucket. Gravity creates increasingly tighter clusters in some of the cells, resulting in a very skewed distribution, approximating a lognormal at late times. The distribution is most skewed at the last snapshot.

filling Peano-Hilbert curve. The PH curve is known to have the best clustering properties among all space filling curves [10], enabling maximally sequential access on the hard drives containing the data for particles close to each other in the simulation volume. Thus indexing along a PH curve makes spatial queries much more efficient.

The Spatial3D library [9] has been developed to use various space-filling curve plugins and define geometric searches. It enables the composition of 3-dimensional shapes from basic geometric primitives; these shapes are accessed via User-Defined types. Searches for points within these shapes are expedited by the space filling curve index, and the shapes exploit the periodic boundary conditions of cosmological simulations. The Spatial3D library is currently being used in the Millennium, MultiDark, and turbulence simulation databases.

For Indra, we use a coarse-grained PH curve and group all particles with the same PH key into the same array. The buckets are leaves of a 6-deep, level oct-tree, with 64 buckets per linear dimension. The total number of the buckets is $64^3$=262,144. So for our test simulation, with $512^3$ particles and a 6-bit PH curve, we can reduce the number of required rows in the Particle table by a factor of 512. This is significant, as there is a 6 byte overhead in SQL Server for each row. This decreases the size of the B-tree indices as well. Additionally, indexing the particles according to a PH curve can greatly speed up spatial queries.

Within an individual PH bucket, we order the particles by their ID (partID). This ensures that the particle at index i in the ID blob will have its position and velocity information at index i in the corresponding position and velocity blobs. While there are other options, like extending the Peano-Hilbert keys down to the level of individual particles, we will see later that this ordering offers certain advantages.

The simulations start from small perturbations on top of a uniform background density, sampled by a discrete array of particles. As a result, the buckets will have an almost uniform initial density, with approximately Gaussian distribution of the bucket counts, with a mean of 512 particles per bucket. However, due to gravitational clustering, the count distribution will soon become very skewed as particles attract one another, and there will be a long tail, corresponding to a hierarchy of gravitationally bound groups and clusters of particles (see Figure 2).

Currently we store both the positions and the velocities as single precision floating point numbers (4 bytes), but it is clear that the data can be represented in a more compact fashion. For each bucket one can store (or calculate from the PH key) the lower left hand corner of the cube describing the bucket. Then each particle's relative position within the bucket can be turned into a 16-bit unsigned integer. As the bucket size is (1024/64) = 16 Mpc, the smallest distance we can represent is $16/2^{16}=2^{-12}$ Mpc. In order to estimate the largest relative distance error, we need to compare this to the size of the box, 1024 Mpc. This dimensionless ratio is $2^{-22}$. The accuracy of this is just 2 times worse than $\varepsilon$, the accuracy of the IEEE floating point representation ($2^{-23}$). As the simulation is using a much larger "softening length" for the calculations of the gravitational force, this truncation error is irrelevant. One can perform a similar compression of the velocities, by scaling them and truncating as 2-byte signed integers. So these simple transformations enable us to save a factor of 2 in the basic storage of the raw data.

## 3. INVERTED INDEX

### 3.1 Backtracking Particles

Another type of query pattern requires the back-tracking of certain particles along their trajectory. Though grouping and indexing the particles according to their Peano-Hilbert index has several advantages, especially when we need to evaluate localized statistics, particle tracking is quite difficult. The brute force option would be to re-organize the data and build a secondary index table sorted by partID and snapnum, also containing the position and velocity. This would effectively double the storage required, adding another PB to the storage requirements. Alternatively, we can search the ID blob in each PH bucket for the relevant IDs, but such queries will be prohibitively time-consuming.

However, these back-tracking queries are typically highly selective, needing position or velocity data on only hundreds or a few thousand of the 1.07 billion particles in the simulations. This problem requires a faster way to search for particles by their ID, without sacrificing our previously described data storage strategies of grouping the particle information within a time step by PH index and storing that grouping as a set of blobs in a single row.

### 3.2 Previous Approaches

In our initial approach to solving the indexing problem, we associated a Bloom filter [1] with each blob of particle IDs at each time step. One of the biggest bottlenecks in finding individual particles based on their IDs is the overhead in unpacking hundreds of thousands of blobs and searching through them, most of which do not contain any data needed to satisfy these highly selective queries. A Bloom filter would allow us to quickly probe each blob without unpacking it to determine whether it contains any relevant data. We would then need to unpack only those blobs that result in a positive match.

However, there were several issues with this strategy. In these particle tracking queries, we are often querying for hundreds or thousands of particles. With a 6 bit PH index, there are 262,144 indices, and thus Bloom filters, for each time step. Probing that many Bloom filters hundreds of times is still prohibitively slow. We then experimented with using Bloom filters to perform set intersection queries. When using Bloom filters with k hash functions, we have a potential common member of two Bloom filters if they have at least k common bits set to 1. We can determine the number of 1 bits the two filters have in common by taking the bitwise AND of the two Bloom Filters. We could therefore take the bitwise AND of the Bloom filter associated with the blob currently being searched and a Bloom filter associated with the list of particles being searched for to determine whether they had a common member. However, while this approach was faster, we lost the nice properties of the low false positive rates of Bloom filters when probed for one value at a time. With this method, the false positive rate was close to 100%, resulting in nearly every intersection query having a potential match. We were getting so many false positives that we were back at the original bottleneck of needing to unpack prohibitively many blobs.

## 3.3 Basic Inverted Index

Rather than rebuilding a transposed index and effectively doubling our storage, we will show how to build an efficient pointer into the existing index. In order to do this, we build first a simple inverted index on the particle data so that we can use the primary keys of our Particle table, then we consider various compression schemes to minimize its size.

Every particle's data (position, velocity) are stored in SqlArray blobs in the Particle table, ordered and indexed by (snapnum, PHkey). The particle's position within this array is given by its "slot," which is a 2 byte unsigned integer. Therefore, each particle's data can be fully identified by a three part address: (snapnum, PHkey, slot).

The index itself is simple in principle. Imagine that we create an additional table in our database for each simulation. This table has four columns: partID, snapnum, PHkey, and slot. The primary key of the table is on the partID and snapnum. The PHkey column contains the PHkey of the particle at the particular snapshot. The slot refers to the offset of that particle's information within the blobs at that PHkey. Because the data are sorted by particle ID within a blob, this slot is consistent for the three blobs in the Particle table, containing partIDs, positions, and velocities.

Storing the slot information alleviates the need to unpack each of the blobs containing relevant particle data and search through them. Using one of the methods of the SqlArray class, we can directly seek to and extract the relevant data within each blob, further speeding up the query.

In the following sections, we show how we built the index, how this information is enough to complete particle tracking queries in seconds, and that we can create the inverted index quickly and with minimal overhead to the simulation data post-processing we already must perform. We also show that we can compress the index to the point where the extra space required for it is acceptable compared to the size of the simulation.

## 3.4 Construction of the Index

Creation of the index takes several steps. The Gadget-2 simulation code outputs flat files at every time step containing the position and velocity of each particle in a special format [14]. When we load the simulation data into the database, these flat files are parsed and transformed into files matching our table schema in the native SqlServer binary format for fast bulk ingest.

The output from the simulation in each snapshot is divided into several files using a high level PH curve, i.e. all particles in a given oct-tree cell are stored in the same file. We transform one file at a time by reading its contents into memory, computing the 6-deep PHkey value from the positions, sorting it by PHkey and then partID, then writing the data back out in the SqlServer native binary format.

These files contain the snapnum, partID, PHkey, as well as the offsets within each PH bucket for each particle, as we write the data to disk. This additional step (on top of the loading the proper base data) adds an approximately 18% overhead to the processing time. We can then use the BCP command line utility to quickly load the files into our database. This is the same technique used to load the transformed particle files. The loading is performed into separate temporary tables for each of the files, using as much parallelism as possible.

However, for our inverted index, we need to transpose these files, where we can place the location data for the same particle collected into a single block, ordered by snapnum. We need the sort order to be partID and then snapnum. This requires an additional sort on the index.

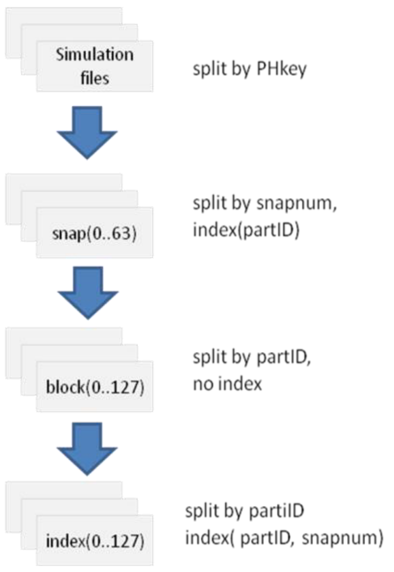

**Figure 3.** Flowchart of the index creation process. The blocking of the data enables efficient memory usage and parallelism.

This sorting of the index is done in the database using a custom merge sort workflow employing multiple steps, coded as stored procedures. We first sort all records within a snapshot by partID. The index for a single snapshot is about 1.5 GB, so this sorting can be done very efficiently in memory. We then split the snapshot into 128 blocks, such that the records from all time steps for the first 1/128th particle IDs are in the first block, the second 1/128th records are in the second block, etc. However, each of these blocks are still sorted by (snapnum,partID). We then re-sort each block in memory (each block is just under 800 MB) by the transposed order of our keys (partID, snapnum) resulting in the final ordering required by our index.

Each of these steps – processing the simulation output, loading into the database, and sorting the index – can be done in parallel. Figure 3 illustrates the index creation process from start to finish. Starting from the flat files that are portioned by PHkey, we can create a partial index for every snapshot independently and thus in parallel. We can then load each of these partial indices into the database in parallel, especially if they are partitioned across multiple disks. Once the data is in the database, we partition it into small blocks that can be sorted in parallel. SQLServer uses the available multiple cores very efficiently. Figure 4 shows the scale-up of the last two portions of this workflow as function of the number of processes employed on an 8-core machine. Each process performed the extraction from the snapshot partial indices to an index block and then sorted the index block by (partID, snapnum) for 8 blocks sequentially. This process did not quite achieve linear scaleup but still performed very well with 8 concurrent processes. One of the next steps in dealing with the Indra data is to build a highly efficient parallel loader for the particle data, fast enough to dynamically load simulation time steps on-demand. Our expectation is to load a single snapshot on the order of one minute. One of the components of this loader will be to efficiently merge partial indices with the index from a newly loaded snapshot to allow for incremental updates to a functioning index (see Section 5).

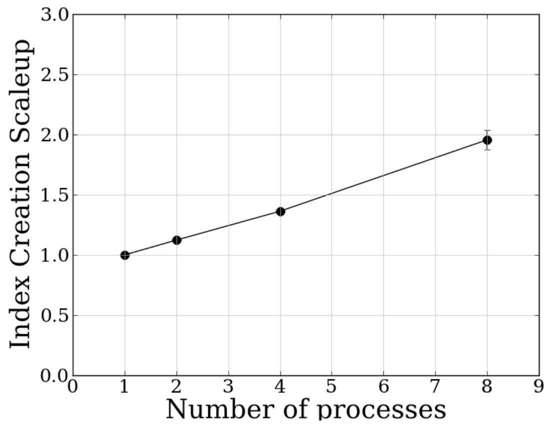

**Figure 4.** Scaleup for splitting the index into blocks by partID and then sorting these blocks. These are the last two steps of the index creation. Each process performed these two operations for 8 blocks sequentially. The base time for one process was 15 seconds.

## 3.5 Particle Tracking Queries Using the Inverted Index

Once the index has been built, it can be used to efficiently perform particle tracking queries. These queries take the following general form[1]:

- Identify the particle IDs of the particles being queried – these are often the member particles of a halo – and place them into a temporary table.
- Join this ID table with the inverted index where the time step of the index matches the desired time step. If querying for the positions at all time steps, no predicate on the time step number is provided. This provides a list of PHkeys and slots needed.
- Consolidate all of the slots for a given PHkey in a given time step into a single array using the SqlArray library, and store this consolidated index data in another temporary table.
- Join the (snapnum, PHkey, slot) table with the Particle table.
- Select the data at the needed slots from each of the blobs.

Figure 5 shows the halo tracking query execution time as a function of the number of particles in the halo. Even the largest halos at over 9000 particles can be tracked in just a few seconds, compared to the several minutes required when not using the index.

One example of a query that requires particle tracking is building a merger tree, which links a given halo to the halos in previous (and subsequent) snapshots by tracing the dynamical evolution as smaller halos combine to form larger halos at later times. Since the halos are identified separately at each snapshot of the simulation, the halo IDs in different snapshots have no relation to each other. Building the merger history of a halo requires comparing the particle lists for halos that have similar locations across consecutive snapshots, and then assigning parent halos to a given halo according to some physical criteria. The speed-up provided by the inverted index will make it feasible to calculate halo merger trees directly in the database.

A simplified version of a merger tree is shown in Figure 6, where just the x positions of a late-time halo are shown for all previous time-steps. In particular you can see a large branch to the right and smaller branch to the left, which would have been a sub-halo that only merged with the larger halo at a time step of around 60. This is an example of the type of data that we can efficiently gather using the inverted index.

Some queries that rely on particle tracking do not depend on the halos. The particle IDs themselves encode information about the Lagrangian positions of the particles, i.e. their initial locations on the 3-dimensional cubic lattice. This information is used, for example, by the ORIGAMI algorithm [6] which measures the morphology of the 'cosmic web' of structures in a simulation by keeping track of whether particles have switched places with respect to their initial configuration. This will require particle tracking for every single particle in the simulation, which will be possible only with the efficiencies that the inverted index provides.

## 4. INDEX COMPRESSION

Here we show how we exploit properties of the underlying physical system being modeled, as well as efficient integer bitwise compression schemes, to significantly reduce the footprint of the index on disk.

The index as described so far is unfeasible for use in petascale simulations such as Indra. A single run of the Indra simulation is just over 2 TB, while the index for a single run of the simulation is about 0.4 TB, or 1/5 the size of a simulation. This means that for a 1 PB suite of simulations, we are also incurring an additional 200 TB needed to store the inverted indices for these simulations. Thus, we turn to a variety of compression techniques to minimize this footprint.

[1] The actual implementations of these queries are as User-Defined Functions in SQL CLR using C#. Using C# makes it easier to interact with the SqlArray library functions.

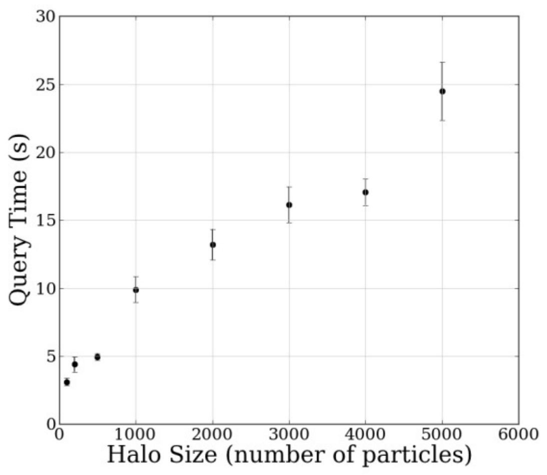

**Figure 5.** Query execution times to find the positions of all particles in a halo at all time steps using the uncompressed inverted index. The times are the average of computing the query for 10 equal-sized halos in the final snapshot. Even the largest halos can be fully tracked in under 30 seconds.

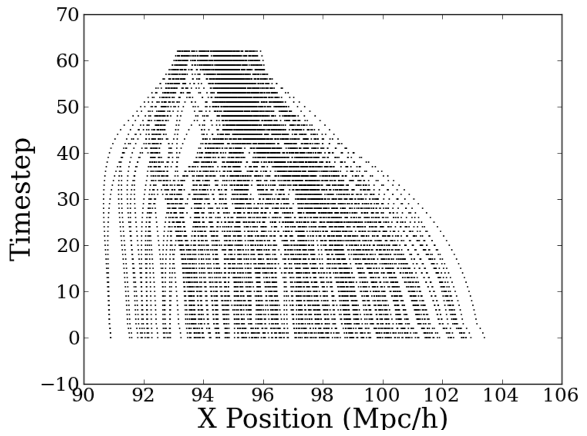

**Figure 6.** The x position of all particles in a 200-particle halo at every timestep. The halo was identified in the final timestep (64). This figure shows the clustering of the particles through time as they are drawn together by gravitational forces. The query to find all of this data took 4 seconds to perform using the inverted index.

## 4.1 Compressing the PHkey Column

Let us consider first accessing the history related to a single particle. To store the relevant 3-part address, we need to create an array for each particle that contains three columns: the snapnum (0.63), the PHkey in each snapshot (18 bits), and the slot (16 bits). As for each particle this table will be exactly 64 rows deep, we can omit storing the snapnum; it will simply map onto the row number of the table. This saves us 64 bytes per particle.

Next, we should look at the statistics over the histories of the particles. Figure 7 displays the probabilities that a particle will cross over to another bucket during the simulation. It turns out that for the bucket size we have chosen, the average number of cross-overs is 1±0.84. This means, that the column containing the PHkeys is incredibly redundant, typically 63 out of the 64 values are identical.

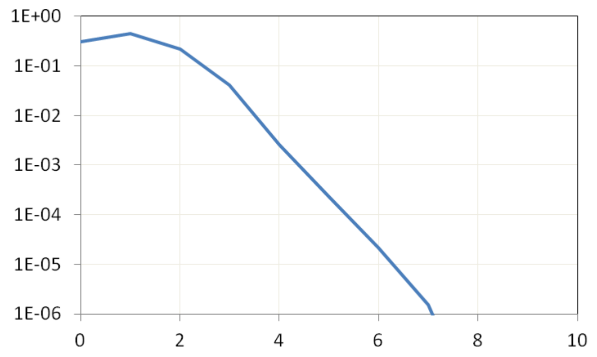

**Figure 7.** Probability of the number of times a particle will cross over to another bucket during the simulation. The x-axis is the number of crossovers.

Furthermore, it is important to note that the particles move continuously and their velocities are small enough that between consecutive snapshots they can only cross into their nearest neighbors. As each bucket has only 26 directly connected neighbors (we have periodic boundary conditions), we can encode the relative displacements as one of 27 distinct values, with 0 designating no crossovers. As a result, we can assume that each column containing the PHkeys can be represented as the initial PHkey, and an array of 5-bit numbers, mostly consisting of zeros. This is effectively a physics-based delta encoding of the PHkeys.

We can compress the data further by noting that every crossover can then be represented in 11 bits, a 6-bit address for the snapnum, and a 5-bit transition code. We also need 6 more bits to store the count of the crossovers, as there are 64 snapshots.

In summary, each PHkey column can be stored in 18+6+11N bits where N is the number of transitions. As the expectation value of N is 1, the typical storage requirement for the PHkey column is 35 bits, compared to 64*32 = 2048 for the raw database column, for an effective compression ratio of 2048/35 = 58.5.

To decode this scheme, we also create a secondary lookup table which stores the correspondence between PHkey neighbor code and actual value for each key. This table is 123.2 MB on disk (for our test simulation), negligible compared to the terabytes needed to store the index for the entire simulation suite, even after being compressed. We can then trace the path of the particle through its adjacent neighbors using this lookup table until we reach the decoded PHkey for the time step being queried. As this table is small, and will be frequently used, it will always remain in hot cache.

## 4.2 Compressing the Slots

The same observations about the physical dynamics of the simulation lead to a compression scheme for the slot portion of the index. PH cells can have thousands of particles in them, and so without any compression we need 2 bytes to store the slot number. But we have already noted that because the particles are relatively slow moving, they don't change PH cells very often. Figure 8 shows the probability distribution of distinct slot numbers for an individual particle. The result is quite depressing at first, as the expectation value is 57, out of the maximum 64.

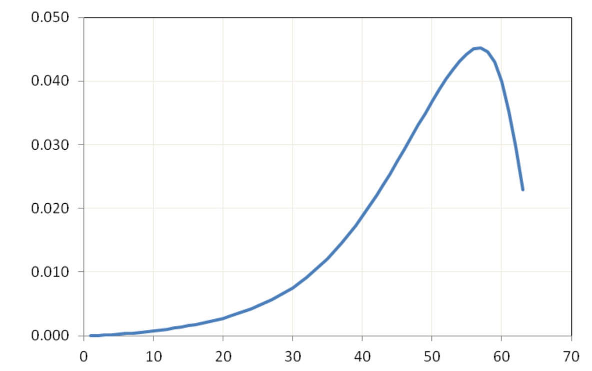

**Figure 8.** The figure shows the probability distribution of the number of distinct slot values for the individual particles. The average is 57 out of the maximum 64.

However, we should realize that in order for the slot number for a particle to change between consecutive snapshots, other particles with a lower ID must either enter or leave the particle's PH cell or the particle must change PH cells. Thus, the change in slot number between adjacent snapshots will tend to be small, as in a given snapshot only 1/64th of the particles are changing buckets. We therefore propose to use first a delta encoding [3] on the slot number, resulting in much smaller numbers to be stored, on the order of 10's rather than 1000's.

Figure 9 displays the probability distribution of the differential slot values (dslot). As the figure shows, the slot deltas are heavily weighted towards small numbers. Furthermore, the distribution is quite skewed towards positive dslot values. First this may seem strange, but this is just a reflection of the skewness present in gravitational clustering. As the distribution of cell counts has a long tail with cells of very high cardinality, these arise as particles

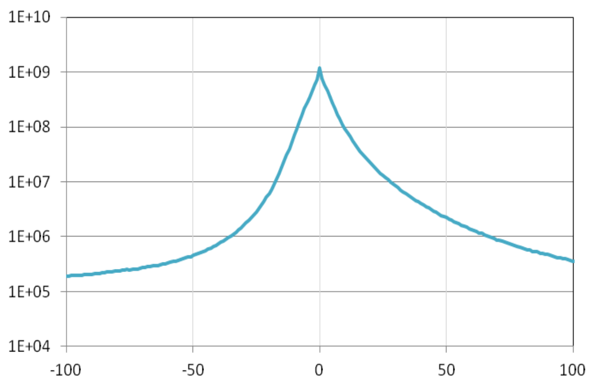

**Figure 9.** The probability distribution of the delta-encoded slot values. Note the skewness of the distribution, and also the sharp peak at zero lag.

from low density adjacent cells are all moving into a high density cell. As a result, there will be only small decreases on the slot values on all the low density cells, while in the small number of high density cells we will generate large dslot values as many new particles are inserted into the ordered lists at every snapshot.

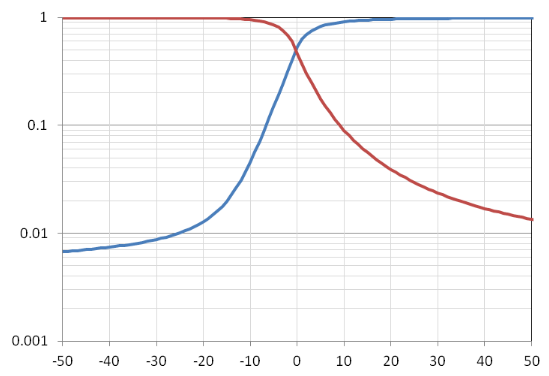

**Figure 10.** The positive and negative cumulative tail distributions of the delta-encoded slot values (dslot).

In order to assess the details of the distribution it is better to plot both the positive and negative tails of the distribution, on a logarithmic scale (see Figure 10).

Using this plot, we can estimate how compressible the dslot values are. We can define a window with a $[d_{min},d_{max}]$ value within which a small number of bits encode the value. The remaining small number of occurrences can be encoded as exceptions taking up a full 16 bits. This is essentially a variant of the Frame-of-Reference (FOR) encoding [3]. We can use N bits to represent the small values, with 0 corresponding to the lower bound,

$$d_{max} = d_{min}+2^N-1, \qquad (1)$$

and we can use the value $d_{min}+2^N$ as marking an exception.

We will use the two-tailed distribution to define a symmetric threshold applied to both the positive and negative tails alike. For the different values of this tail probability we will get both a positive and negative limit for the exceptions. For N bits, these values should be $2^N$-1 apart.

For example, for N = 4, we should select a threshold for the tail probability to be 0.100, i.e. 20% of the slots will be exceptions. However, the remaining 80% can be then coded in 4 bits. The table below shows the respective thresholds for N = 4, 5 and 6. If the tail probability is p, for a given N, then the effective number of bits is given by

$$B = N + 2p\cdot 16. \qquad (2)$$

The summary of these different N values is shown in Table 1. It is clear that there is an optimum: N = 5 gives the best compression for the dslot column at 6.28 bits per slot.

**Table 1**. Summary of different thresholds for our FORDelta compression scheme

| bits | range | tail probability | effective bits | compression ratio |
|---|---|---|---|---|
| 4 | 15 | 0.100 | 7.20 | 2.50 |
| 5 | 31 | 0.040 | 6.28 | 2.72 |
| 6 | 63 | 0.015 | 6.48 | 2.54 |

## 5. CONCLUSION

The Indra suite of cosmological N-body simulations will provide excellent statistics of the large scale features in the distribution of dark matter while at the same time resolving the nonlinear evolution of structure. It will present the best effort to-date toward overcoming the cosmic variance limit on the uncertainties we can derive for large-scale structure measurements of cosmological parameters. When complete, Indra will be made available to the community through a state of the art database infrastructure.

The Indra simulation data will consist of $1024^3$ particles per snapshot, plus halo catalogs and the complex Fourier modes of the density field, for 64 snapshots and 512 individual simulations, resulting in 35 trillion particles and 1.15 petabytes in total.

The database will make use of the SqlArray library by grouping the particles into chunks according to the space-filling Peano-Hilbert curve, greatly reducing the number of required rows and thus the SQL Server overhead. Though grouping and indexing the particles according to their PH index has several advantages, especially when we need to evaluate localized statistics, particle tracking is quite difficult.

In this paper we have presented a strategy for creating an efficient inverted index for these data that speeds up particle tracking queries so that they complete in seconds. The index can exploit the physical characteristics of the particle motion in cosmological simulations to implement a compression scheme on this index.

The two part address of our inverted index can be heavily compressed. The first part, describing the PHkey is reduced by a factor of 58.5. The second part, describing the slots, can be also compressed, using a combination of Delta and Frame-of-Reference encoding. The compression ratio for the second part of the address is 2.72. The resulting mean compression ratio for the whole index is 202/22.5 = 8.97, reducing its footprint to 22.5TB (see Table 2).

**Table 2**. Summary of the Indra simulation cardinalities and data sizes, including the raw and compressed inverted indices. The number of particles is given in billions.

| | | | | Index [TB] | |
|---|---|---|---|---|---|
| sims | snaps | part [B] | data [TB] | raw | comp |
| 1 | 1 | 1.1 | 0.04 | | |
| 1 | 64 | 68.5 | 2.24 | 0.40 | 0.044 |
| 80 | 64 | 5478.4 | 179.20 | 31.60 | 3.520 |
| 512 | 64 | 35061.8 | 1146.88 | 202.24 | 22.528 |

One of the challenges with the inverted index as presented here is that we must create the index for an entire simulation at once in order to get good compression performance. Our merge-sort workflow to sort the index also relies on creating the complete index at once. We plan on exploring ways to efficiently generate and compress a partial index on only some of the snapshots, as well as ways to efficiently merge these partial indexes together. For example, faster compression and decompression schemes could make the process of updating the index faster.

We are also in the process of building a fast parallel loader to load simulation data into the database quickly. We believe we will be able to load a single snapshot into the database on the order of a minute by using a server with a large number of cores and storing the raw data on SSDs for faster I/O throughput. If we reach these speeds we can dynamically load in snapshots on demand, rather than needing to load an entire simulation into the database at once. Being able to incrementally update the inverted index one snapshot at time would make this dynamic loader even more effective.

## 6. ACKNOWLEDGMENTS

This research is partly funded by the Gordon and Betty Moore Foundation through Grant GBMF#554.02 to the Johns Hopkins University. The research was also supported by the NSF grant OIA-1124403.

## 7. REFERENCES

[1] Bloom, B. H. 1970. Space/Time Trade-offs in Hash Coding with Allowable Errors. Commun. ACM 13, 7 (July 1970), 422-426. DOI=http://doi.acm.org/10.1145/362686.362692

[2] Davis, M, Efstathiou, G., Frenk, C. S., and White, S. D. M. 1985. The evolution of large-scale structure in a universe dominated by cold dark matter. Astrophys. J. 292, 371-394.

[3] Delbru, R., Campinas, S., Samp, K., and Tummarello, G. 2010. Adaptive Frame of Reference for Compressing Inverted Lists. DERI Technical Report, 2010-12-16.

[4] Dobos, L., Szalay, A. S., Blakeley, J., Falck, B., Budavári, T., and Csabai, I. 2012. An Array Library for Microsoft SQL Server with Astrophysical Applications. Astr. Soc. P. XXI 461, 323-327.

[5] Eisenstein, D. J., Hu, W., Silk, J., and Szalay, A. S. 1998. Can Baryonic Features Produce the Observed 100 H -1 MPC Clustering? Astrophys. J. Lett. 494, L1-L4.

[6] Falck, B. L., Neyrinck, M. C., and Szalay, A. S. 2012. ORIGAMI: Delineating Halos Using Phase-Space Folds. Astrophys. J. 754, 126-136.

[7] Jenkins, A. 2010. Second-order Lagrangian perturbation theory initial conditions for resimulations. Mon. Not. R. Astron. Soc. 403, 1859-1872.

[8] Lemson, G. and the Virgo Consortium 2006. Halo and Galaxy Formation Histories from the Millennium Simulation: Public release of a VO-oriented and SQL-queryable database for studying the evolution of galaxies in the LambdaCDM cosmogony. e-print arXiv:astro-ph/0608019.

[9] Lemson, G., Budavari, T., and Szalay, A. S. 2011. Implementing a General Spatial Indexing Library for Relational Databases of Large Numerical Simulations. Lect. Notes Comput. Sc. 6809, 509-526.

[10] Moon, B., Jagadish, H. V., Faloutsos, C., and Saltz, J.H. 1996. Analysis of the clustering properties of the Hilbert space-filling curve. IEEE Transactions on Knowledge and Data Engineering 13, 2001.

[11] Perlman, E., Burns, R., Li, Y., and Meneveau, C. 2007. Data Exploration of Turbulence Simulations using a Database Cluster. Supercomp. Proc., ACM, IEEE, 23.

[12] Riebe, K., Partl, A. M., Enke, H., Forero-Romero, J., Gottloeber, S., Klypin, A., Lemson, G., Prada, F., Primack, J. R., Steinmetz, M., and Turchaninov, V. 2011. The MultiDark Database: Release of the Bolshoi and MultiDark Cosmological Simulations. e-print arXiv:1109.0003.

[13] Samet, H. 2006. Foundations of Multidimensional and Metric Data Structures. Morgan-Kauffmann, San Francisco, CA.

[14] Springel, V. 2005. The cosmological simulation code GADGET-2. Mon. Not. R. Astron. Soc. 364, 1105-1134.

[15] Wu, K., Ahern, S., Bethel, E. W., Chen, J., Childs, H., Cormier-Michel, E., Geddes, C., Gu, J., Hagen, H., Hamann, B., Koegler, W., Lauret J., Meredith, J., Messmer, P., Otoo, E., Perevoztchikov, V., Poskanzer, A., Prabhat, Rübel O., Shoshani, A., Sim, A., Stockinger, K., Weber, G., Zhang, W-M. 2009. FastBit: Interactively Searching Massive Data. J Phys Conf Ser 180, 1 (2009), 012053. DOI=http://dx.doi.org/10.1088/1742-6596/180/1/012053

[16] Yan, H., Ding, S., and Suel, T. 2009. Inverted Index Compression and Query Processing with Optimized Document Ordering. Proceedings of the 18th international conference on World wide web. ACM, New York, NY, 401-410. DOI=http://doi.acm.org/10.1145/1526709.1526764