

Stephen Channell

Cepheis | lONDON

**Hiperspace**

**High Perforamance Space**

Contents

[Introduction 5](#_Toc155040846)

[Performance 6](#_Toc155040847)

[Mutability 7](#_Toc155040848)

[Schema evolution 8](#_Toc155040849)

[Use Cases 9](#_Toc155040850)

[FRTB 9](#_Toc155040851)

[Artificial Intelligence 10](#_Toc155040852)

[Graph View 10](#_Toc155040853)

[Document Store 11](#_Toc155040854)

[Web Applications 12](#_Toc155040855)

[Time series 12](#_Toc155040856)

[Durable memory 13](#_Toc155040857)

[Architecture 14](#_Toc155040858)

[HiperSpace 17](#_Toc155040859)

[Generation Space 18](#_Toc155040860)

[SubSpace 18](#_Toc155040861)

[SetSpace 19](#_Toc155040862)

[RefSet 20](#_Toc155040863)

[RefSingle 20](#_Toc155040864)

[KeyRef 20](#_Toc155040865)

[KeyPath 21](#_Toc155040866)

[IndexPath 22](#_Toc155040867)

[HiLang 23](#_Toc155040868)

[Statements 24](#_Toc155040869)

[Definition 25](#_Toc155040870)

[Documentation 26](#_Toc155040871)

[Property 27](#_Toc155040872)

[Name 28](#_Toc155040873)

[Reference 28](#_Toc155040874)

[Inheritance 29](#_Toc155040875)

[Views 29](#_Toc155040876)

[Keys 30](#_Toc155040877)

[Values 31](#_Toc155040878)

[Extents 32](#_Toc155040879)

[Enums 33](#_Toc155040880)

[Directive 34](#_Toc155040881)

[Expression 35](#_Toc155040882)

[Boolean 36](#_Toc155040883)

[Hash 36](#_Toc155040884)

[Prelude 37](#_Toc155040885)

[Types 37](#_Toc155040886)

[Collections 38](#_Toc155040887)

[Functions 38](#_Toc155040888)

[Graph Views 39](#_Toc155040889)

[Index 42](#_Toc155040890)

# Introduction

Hiperspace is an object storage technology that uses key addressable store to expand the quantity of information directly addressable by an application without duplication or explicit serialization. Elements that are not already in main memory are loaded transparently when needed, and released when they are no longer required.

HiLang is the data domain specific language that generates code to bridge the domain elements to keys and values loaded stored in Hiperspace. Conceptually it is the like an object/relational mapping technology but for key-stores (using modified Protobuf[[1]](#endnote-2) serialization) with fast compiled access paths.

Mapping domain entities directly to key-stores significantly reduces latency, avoiding the need to pre-load related entities.

Separate message formats are not needed to pass elements from a server to a client - transparent reference loading stops when the SubSpace is closed.

Separate error-log formats are not needed because referential integrity and validation constraints are enabled by when opening a SubSpace – error-logging spaces are opened without constraints.

The runtime is pure open-source and can be deployed without licenses.

## Performance

Performance benchmarks[[2]](#endnote-3) against the leading server databases have shown four-fold improvement traversing a graph of hundreds of thousands of observations, and ten-fold improvement for a random update.

Elements are referenced using:

* Full-key  
  keys/values are stored directly in hiperspace with efficient lookup.
* Partial-key  
  Sets of dependent segments can be loaded as a collection using the first part[[3]](#endnote-4) of a key.
* Index  
  Where the path from one element to another does not involve the key, index entries are added to efficiently access all matching elements.

A simple rule-based optimizer ensures that any request for an element uses a key or index for access if one is available.

Keys do not need to be a flat structure like a relational table but can be composed of arbitrary structures of Values, key and lists. Keys can be up to 8Mb and Values can be up to 2Gb, but large keys are not recommended.

As High-performance Memory devices mature[[4]](#endnote-5) higher performance will be possible using shared pooled memory and accelerated key search.

## Mutability

Elements within a hiperspace are immutable (but versioned) once bound to a space, no need to synchronize with changes that might also be performed elsewhere. Elements that do need to change are efficiently versioned so that thousands of versions can be stored without impacting performance.

All modern remote database management systems us this technique to durable recovery in the event of failure, but normally hide the versions behind distributed locking in database logs. Application can additionally use temporal tables[[5]](#endnote-6) but special queries are needed for access.

Avoiding locking yields most of the performance advantages of hiperspace. The complete history of a single Element can be retrieved from backing store, or a SubSpace can be opened for a point-in-time view of an entire hiperspace. When opened with an “AsAt” parameter, newer versions are skipped over.

Mutable Elements cannot be deleted – where the domain requires the notion of deletion, Hiperspace achieves this with tombstoning. Adding a tombstone attribute to each Element (e.g. “Deleted”) and a horizon filter to a SubSpace hides any element with a tombstone, but is still visible when an as-at SubSpace is opened.

## Schema evolution

Immutability provides significant advantages for performance and concurrency but does have one problem: what to do when new attributes are discovered, and old attributes dropped or changed.

Hiperspace allows the logical schema to evolve without the need to re-build stored values using four concepts:

* Aspects  
  Optional information or added later are stored separately form the main Element but transparently read as if they were properties of the Element. Aspects can be added to design later and will be invisible to earlier versions of code. Aspects do not have keys since they will inherit the key from their owner. A typical aspect is an approval status, cost, or risk-value-adjustment.
* Segments  
  Segments are sets of Elements that have a key-part in addition to the key inherited from the owner. Because segments are stored separately from the main Element, they can be added later. A typical Segment is the Transaction associated with an Account, Customer or Trading-book.
* Alias  
  Each key field or value attribute has an optional numeric #alias for storage (it is generated if not provided) – leaving a gap in future schema versions will cause a value to be ignored when loaded, adding a value will be returned as missing for older Elements. Aliases is not recommended for evolving a schema unless it was caused by a genuine design flaw, since values cannot be retrospectively added, and information will be lost (but not save space).
* Views  
  Views are a feature of Hiperspace to project a common view from several different entities in a domain that have different shape. Adding the latest vision of ESG[[6]](#endnote-7) metrics can be achieved by adding a view, an updated version of entity, and adding aspects to the older version. Hiperspace views are like SQL ‘UNION ALL VIEW’ and access each realization that provides the view. Data can be viewed as a graph (node & edge) using pre-defined views.

Care needs to be taken when evolving a schema not to change the definition accidentally, since it may result in access being blocked.

# Use Cases

## FRTB

Fundamental Review of the Trading Book[[7]](#endnote-8) is a challenging regulation because it requires historical retention of information for back testing of model changes to verify that risk forecasts are consistent with actual data. The standard approach is to warehouse each daily datasets separately in a data lake[[8]](#endnote-9).

Hiperspace solves the problem of historical views with native versioning of elements, accessing As-At is as simple as providing the parameter when opening the SubSpace. Tests with tens of thousands of revisions do not significantly impact performance because the latest version is always found first. Using segments and aspects the version history can be confined to only the parts of a model that changes – if only the price or rate changes, only that part needs to be versioned.

Schema evolution allows older versions of financial models to view data exactly as it was when created, while newer versions see any additional aspects added later.

Whole portfolios can be added again to Hiperspace, but only parts that have changed are stored as new revisions.

With unlimited versions, support of arbitrary complex nested objects up to 2Gb in size and transparent partitioning, data volumes do not grow exponentially. It is practical to use a single store for current and historical data.

## Artificial Intelligence

Hiperspace does not offer or use AI, but AI has introduced a new dimension to the problem of complex data because it does not update source reference data but requires that it be read millions of times more often. Hiperspace has been designed to take advantage of the new memory technologies being built for scale. Hiperspace was designed for generations of immutable historical data that can be replicated for parallel search by enormous numbers of agents.

Hiperspace uses relatively simple key/value stores like RocksDB that are specifically designed for efficient storage in shared memory pools and SSD. Log structure merge storage always uses a fraction of durable space than traditional databases because they do not need to provide padding space for updates. In a performance test of Hiperspace against SQL/Server 2019, Hiperspace used 20% of the space and took 20% of the time to search.

## Graph View

Every data model can be decomposed into Nodes of durable facts and Edges connecting them together. Hiperspace addresses the graph view need by breaking the problem into three parts:

* Presenting the domain model elements as either a Node, an Edge, or a combination of both[[9]](#endnote-10). In Hiperspace a Node is a view that an element projects, while the set of Nodes includes all element types that project the Node view and all instances of each type.
* Hiperspace transparently traverses from element to element as if they were all in memory.
* Functional programing languages like F# can be used to search over a graph of objects without translating the logic into a graph database specific language.

For the highest performance of deep recursive search, it is vastly more efficient to map the population of edges into memory and use a GPGPU to recursively search in parallel.

## Document Store

Hiperspace support for arbitrary complex models and transparent navigation between elements makes it a good match for document-oriented data models where the schema is already understood – the only difference is:

1. The identity of the element must be known.
2. The identity must be part of the definition – it is rare for a document not to include the moniker by which it is known.

The HiLang domain language is almost identical to the equivalent typescript definition, but allows parts (e.g. Address) to be stored as a value with the element or separately with transparent lookup when needed.

|  |  |  |
| --- | --- | --- |
| TypeScript | Protobuf | HiLang |
| class Customer = {  Id : number;  Name : string;  Address : Address; } | message Customer {  int32 Id = 1;  string Name = 2;  Address Address = 3;} | entity Customer  ( Id : Int32 #1)  { Name : String #2,  Address : Address #3}; |

With Hiperspace, the decision whether Address is included in the state of the Customer is determined by the definition of Address, avoiding the need to choose whether to include a key-identifier or value when modelling Customer.

The same schematic translation applies to the XML Schema, but for excluded for brevity.

See [/tree/master/examples/Hiperspace.Sparx](https://github.com/channell/Hiperspace/tree/master/examples/Hiperspace.Sparx) for an example of a large-scale document problem (UML document of a complex library), where structure is modelled directly.

## Web Applications

The latest generation of web applications use web-assembly[[10]](#endnote-11) to deploy application logical directly within a browser without conversion to JavaScript. Hiperspace can be used in client for transparent load/store through a private API that uses a web-tier Hiperspace bridge.

## Time series

Hiperspace is well suited to Time-series requirements because it imposes a minimal overhead to write and is integrated with RocksDB which provides the highest performance for write intensive workloads.

Four features enable Hiperspace to be used with streaming price tick data:

1. When SetSpace.Bind() is used with ‘cache = false’ elements are not cached for re-read and released from memory.
2. All SetSpace functions have an Asynchronous version that does not block the data producer.
3. Functions are implemented in the host language ( see [hours](https://github.com/channell/Hiperspace/blob/b3e608482e95e961c20bda8caca6a2b008a3c31e/examples/Plan/Plan/Calc.cs#L10) function, and sample [usage](https://github.com/channell/Hiperspace/blob/b3e608482e95e961c20bda8caca6a2b008a3c31e/examples/Plan/Plan/Plan.hilang#L3) in the Plan example[[11]](#endnote-12)).
4. Scoped SubSpaces release cached data as when they are closed.

For very large objects like volatility surfaces snappy compression[[12]](#endnote-13) (with Hiperspace.Rocks) provides very high throughput with low compression overhead.

## Durable memory

Durable memory was a concept outlined by Intel for its Optane[[13]](#endnote-14) memory technology but faltered with the need for applications and operating systems to explicitly support it. Optane has been superseded by CXL memory (backed by SSD) that allows large memory pools to be shared by several servers and recovered even when servers fail.

For durable memory, Hiperspace addresses the API need and delegates the technology integration to a Hiperspace adaptor. The current generation of CXL memory focuses on access, but development is being undertaken by storage vendors to generalize the ‘key’ from being purely numeric to a broader universal identifier, with content addressable filtering.

The concept was outlined in the early 1970’s with IBM’s “Future Systems Project”[[14]](#endnote-15) and ICL’s Content Addressable File-Store[[15]](#endnote-16), but lost favor with the development of more capable operating systems. It would have remained a footnote in the history of computing, were it not for the advent of GPGPU technology that can be embedded in storage devices, and the economic driver to support large AI workloads.

A technology that focuses on immutable objects might seem an odd fit for durable memory, but two factors make it applicable:

1. Hiperspace removes the need to focus on a small working-set of memory, allowing objects that are no longer referenced to migrate to slower store (like an operating systems swap-file), but usable for cloud services where swap-files are not available.
2. Where high-reliability/fast-recovery is required, it is common to checkpoint current state anyway.
3. For simulations it is highly beneficial to restart a simulation at a point where an algorithm stopped.

The default RocksDB driver is designed for high performance streaming of updates with non-blocking writes.

# Architecture

The Hiperspace runtime is GPL[[16]](#endnote-17) open-source and can be used without additional licenses. For dotnet, there are initially two Nuget packages for Hiperspace[[17]](#endnote-18) and Hiperspace.Rocks[[18]](#endnote-19) (adaptor for RocksDB[[19]](#endnote-20)). Additional adapters can be built for specific requirements and will be added over time to meet demand.

RocksDB is maintained by Facebook and builds on the work of Google developing LevelDB. It is an open-source key/value store with derivatives tuned for specific workloads, including Speedb[[20]](#endnote-21) and Samsung ‘Processing In Memory (PIM)’[[21]](#endnote-22). PIM is particularly interesting because it could offload the work of RocksDB to a GPGPU embedded in the CXL memory controller for High-Bandwidth Memory[[22]](#endnote-23)

The initial Hiperspace.Rocks adaptor uses local file-system access but is a trivial change to use [RockDB-Cloud](https://github.com/rockset/rocksdb-cloud)[[23]](#endnote-24) for concurrent access from several servers. Adaptors for other Key stores are simple.

A diagram of a computer

Description automatically generated

All Elements in a Hiperspace derive from classes exposed by the Hiperspace assembly to enable them to be composed and linked transparently. The base classes as kept as simple as possible to avoid the need to refactor code over time and implemented as value types wherever possible and abstract base classes otherwise.

At runtime Hiperspace uses opaque byte objects for load/store of value-types that represent the state of domain Elements. The Domain SubSpace, key and indexes paths are generated (ahead of time) in domain-specific assemblies using the HiLang compiler. HiLang is integrated into the build environment for dotnet. Other targets will be added to meet demand.

Application Designers do not need to know the internals of how hiperspace works, all required code is generated by HiLang from the Domain specific language or Sparx Enterprise Architect[[24]](#endnote-25) UML modelling tool.

This section provides an overview of the Hiperspace design but note that performance tests have demonstrated high performance relative to an equivalent relational database.

A diagram of a product

Description automatically generated

Hiperspace Elements are domain objects that are immutable once bound to a SubSpace. Within each Element the state is split into internal Key and Value structs that are passed at runtime to Hiperspace to serialize to and deserialize from backing store. Elements are used directly without proxies[[25]](#endnote-26) – when not bound to a SubSpace they are as small as plain value object.

References from one element to another are supported by a KeyRef that contains a copy of the referenced elements Key, and optionally a *cached* reference to the Element object. When a Value is read, the KeyRef uses the SetSpace reference to lookup the element using the key (the SetSpace will look up missing values using its SubSpace reference). Further reference to Element will then use the cached value[[26]](#endnote-27). KeyRef is extremely fast because there is no query generation, and often already cached in a SetSpace.

For collections RefSet has a functor to create a search template for key/index lookup of related elements within the SetSpace of the referenced type. The domain SetSpace uses the template to search by KeyPath or IndexPath for related elements. Once retrieved, the results are cached for future reference. RefSet does not use a runtime query and does not duplicate objects – there is only ever one copy of any Element within a SetSpace, or SubSpace.

*What about network transfer? When SubSpace is closed, all lazy loading stops. Using a local scope domain SubSpace (*[*example*](https://github.com/channell/Hiperspace/blob/master/examples/util/export.fs#L195)*) prevents lazy loading of references not already cached.*

The following sections outline the role of these common base types.

## HiperSpace

A diagram of a computer

Description automatically generated

Hiperspace can contain every element for all time but the domain SubSpace we use for access apply filters for the time horizon and horizon filters for validation and context. Hiperspace.Rocks is the first example adaptor that bridges HiperSpace to storage technology, additional adaptors (e.g. Redis) are transparent to HiperSpace users.

### Generation Space

GenerationSpace can be used to partition the current working set from historical versions transparently. Scenarios where you’d choose to do this include:

* Partitioning of daily data form sealed historic versions that do not change after close of day.
* Highly parallel update where a shared cache (e.g. Redis) is used for updates.
* Distributed history, where read-only data is duplicated on each server for performance.
* Batch processing where an entire day’s data may be reviewed as a whole and merged once approved.

### SubSpace

SubSpace is the base space that domain spaces derive from. It adds AsAt date parameter for temporal time filtering and Horizon filters for domain specific constraints. Any number of Subspaces can be opened for a an underlying Hiperspace to compare time horizons or context specific values. SubSpace provides basic query infrastructure but is currently limited to queries that do not require reflection or code generation.

SubSpace is the Hiperspace equivalent of DbContext in dotnet Entity Framework or Entity Manager in Java Hibernate.

Applications can subscribe to HiperSpace events for telemetry and other purposes.

## SetSpace

A screenshot of a computer screen

Description automatically generated SubSpaces contain several SetSpaces, one for each Element type stored in Hiperspace, in the same way that Entity-Framework has a DbContext and several DbSet<> collections. The base SubSpace includes two SetSpace references for graph Nodes, and Edges, with implementation in each domain[[27]](#endnote-28). All SetSpaces are created when a domain Space is created and extract Horizon predicates that apply to its content.

SetSpace ensures that only one instance of each Element is created within SubSpace.

A single element can be retrieved from a SetSpaces using the Get() function and a range of elements using the Find() function or using a LINQ[[28]](#endnote-29) expression for SQL like query.

### RefSet

A RefSet<> is a subset of the SetSpace for foreign key references from one element to another. They are created with helper functions to load related elements as needed. References from one Element to a set of relations in the HiLang specification are used to derive the indexes for efficient lookup. An element can be added to a RefSet<> at any time since they are stored separately in Hiperspace. For bidirectional relations (e.g. Customer has a RefSet of Accounts and Account has a RefSet of Customers) Refresh() can be used to reference changes to the underlying content.

### RefSingle

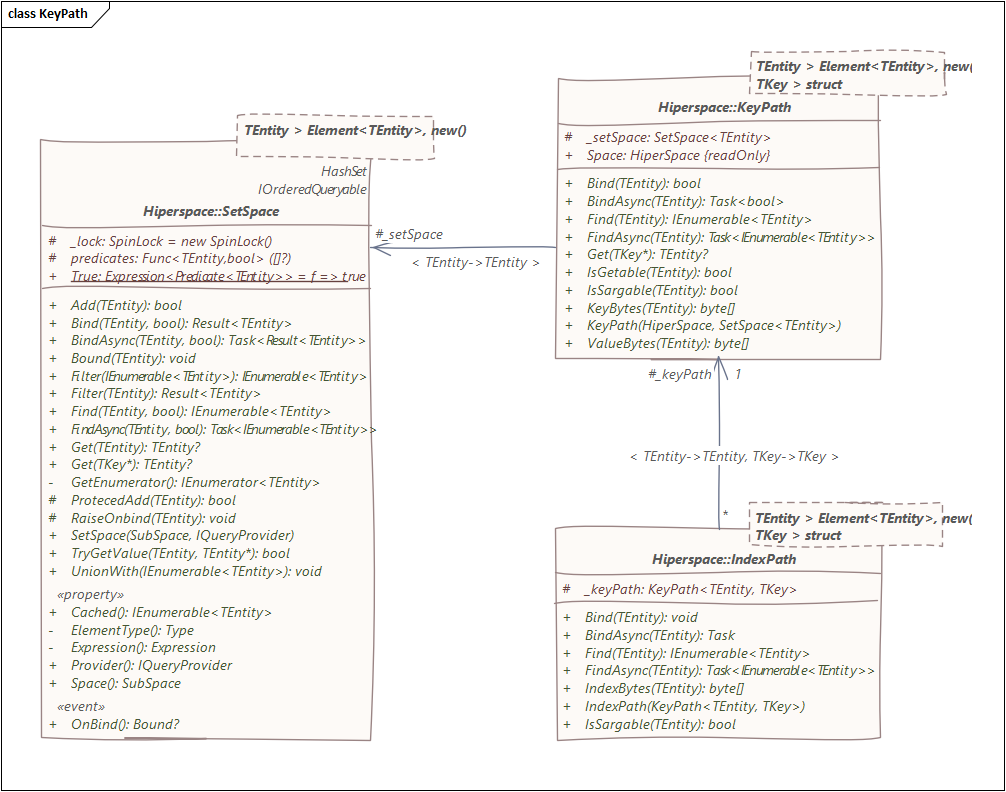
RefSingle<> provides the same function as a RefSet where there is only one related aspect for the element, and the aspect inherits the key of its owner. RefSingle<> is created with functors to retrieve Aspects as needed. Information (such as Value at Risk (VAR)) that is not known when an element is created, can be added as an aspect later.

### KeyRef

KeyRef<> serves as a reference from one element to another in Hiperspace, enabling lazy retrieval from Hiperspace. When creating a KeyRef, a key and optional value are specified, and if the referenced element is not already cached, it will be loaded transparently. Each Hilang element includes a "self" property and cast operators that facilitate the seamless assignment of a value as a reference within an element.

KeyRefs utilize the SetSpace property for performing lookups as necessary. This reference is cleared when SubSpace is closed, ensuring that serialization only includes the relevant parts required by the user.

## KeyPath



Hilang automatically generates domain specific SetSpace<> implementations for each Element in Hiperspace. These SetSpace implementations utilize a KeyPath or optional IndexPaths for loading elements, employing a rule-based optimizer that selects the fastest path to retrieve elements.

Each SetSpace implementation contains a KeyPath, which divides the Key and Value into byte arrays used for efficient loading from Hiperspace. In cases where the template includes a value (such as a reference to a parent), the KeyPath might not be considered Sargable[[29]](#endnote-30) and an IndexPath will be used instead.

*Hiperspace adheres to the principle of "the key, the whole key, and nothing but the key" for sargable search, ensuring optimal search operations. Even if only the owner part of a key is provided, Hiperspace still maintains sargability.*

*To achieve this, Hiperspace reorders the bytes within the binary key value, ensuring that missing key-parts are included in the search parameters. This allows efficient search even when some key parts are not explicitly provided.*

### IndexPath

When searching for items through an IndexPath, the corresponding key is used to load the associated value using the KeyPath. This ensures efficient retrieval of values that are linked to foreign references in Hiperspace.

In Hiperspace, an element's SetSpace can have multiple IndexPaths. These IndexPaths are automatically generated by HiLang for foreign references that are not based on the elements' key. Each IndexPath contains the members of the foreign reference along with a copy of the element's key. When an item is found through an IndexPath, the key is used to load the corresponding value using the KeyPath mechanism. This approach enables efficient retrieval of values tied to foreign references in Hiperspace.

*Indexes take advantage of the byte ordering to ensure that a missing key-part in a search (i.e. the Element key) matches all elements that share this index.*

The following HiLang creates a hierarchy of parties where the Parent reference is used to load a collection of children. Hilang generates an index to load using the parent key.

entity Party

( Id : Int32 )

{ Parent : Party }

[ Children : Party ( Parent = self ) ];

# HiLang

HiLang is a data domain specific language, that translates the domain model ahead-of-time into code that matches the domain model, but uses the runtime capabilities of Hiperspace to store, retrieve and search elements in the store.

The history of computer science is littered with domain specific languages that have been devised at various times to provide capabilities that are not currently supported, most of them have failed or been superseded. Notable exceptions include SQL and protobuf because they have focused on providing the minimum capability for the specific technology domain, allowing integration with tools and reuse in several settings; HiLang attempts to follow this pattern with the minimal additions that are required with excellent integration with tools.

The example HiLang model for integration with Sparx Enterprise Architect[[30]](#endnote-31) demonstrates that 1408 lines of domain definition generates 40,360 of clean compiled C# code, that would take considerably longer to write by hand. When considering whether a domain specific language justifies the effort of using it, it is important to focus on the effort that would otherwise be required to achieve the same outcome.

The HiLang tool[[31]](#endnote-32) integrates directly with the C# Rosyln compiler to give designers and developers an interactive experience highlighting any syntax errors.

The (mini) domain specific language is detailed below using syntax diagrams[[32]](#endnote-33)

## Statements



HiLang statements are separated by semi-colons “;”. There can be any number of statements in a “.hilang” file and do not need to be in a specific order, they will be resolved after the source file has been parsed to AST[[33]](#endnote-34). Each statement can be a definition, directive or enum.

Comments in a “.hilang” file are delimited with “/\*” and “\*/” and do not appear in the generated code.

Syntax diagrams are read left to right following arrows. The diagram above starts on the left followed by either a definition, directive or enum, followed by a ‘;’. After the ‘;’ the syntax either ends or loops around for another definition, directive or enum.

### Definition



Definitions describe the elements of a HiLang model, with different keyworks used to describe the way that each element is stored in Hiperspace. The definition must include the storage class of the element and name, but all other parts are optional.

|  |  |
| --- | --- |
| Type | Meaning |
| type | The element is externally defined, and only the reference is included in the HiLang file. “type” does not have keys, values, or extensions |
| value | “value” is stored in the element that references it. Values can include any number of values, or references as needed to model the domain  “value Fixing(At:DateTime){Rate:Decimal};” can be included as “Fixings:Set<Fixing>” within another element. |
| entity | The element will be stored in Hiperspace using its key with a “SetSpace<entity>” included in the domain space for lookup. Any references to the entity in other elements will include just the key, with lookup within Hiperspace as needed.  Entities must have a key, but values are optional. |
| segment | Segments are dependent types of the entity that is extended with them, and inherit a reference to the entity that owns them, so “segment Payment … {Amount: Decimal};” for Account and Book will be generated as “AccountPayment” and “BookPayment”, each concrete definition including “owner” key to refer to the Account or Book that owns them.  As with entities, segments have a “SetSpace<segment>” reference in the domain space for lookup directly (using their owner and or through the owning element. Segments can also have segments, each of which inherits the owner reference to the elements that are extended with them.  Segments can be added to Hiperspace without reference to the owning element (for immutable (i.e. not Versioned) entities this is the only way to add them). |
| aspect | Aspects are like segments but must not have keys since they are fully identified by the element that owns them. In the generated code, they are referenced like an attribute since we know there will only be one of them.  Aspects are especially useful for information that is not known when an entity is created, or that can change later. Examples include closing price, or business-status.  As with entities and segments, aspects can be referenced directly in Hiperspace. |
| view | Views are not stored directly within HiperSpace but do have a SetSpace<view> for direct lookup. Lookup of a view in a domain space will enumerate every element that includes the view in its definition. The view type is used for “Node” and “Edge” to view the SubSpace as if it was a graph database. |

### Documentation



Documentation is delimited by double-quotes and can include any number of lines, HTML documentation or markdown text. The documentation is added to the generated code as summary comments that are included in tool-tip information when the domain model is used by a developer.

They are distinct from “/\* … \*/” comments and can be applied to any element, field, value or extent.

When the XMI[[34]](#endnote-35) export/import is selected by a command line tool, the documentation is synchronized with a modelling tool.

### Property



There can be a single property, or several properties separated by commas. Each property can optionally include parameters. There can be a single parameter, or several separated by commas.

The most common property is @Versioned, that indicates that an unlimited number of versions can be stored within a Hiperspace together with an “AsAt” timestamp. Properties included in the generated code.

For example, the following property indicates that the element includes the Entity Framework annotation and is versioned in Hiperspace.

@Table ( "t\_object" ), Versioned

// is added to C# source as

[Table ( "t\_object" ), Versioned]

### Name



Names are normally just a reference, but can be generic with {“<”, “>”} separating the type parameters. Entities cannot have generic parameters, but all other generic types will be expanded during compilation (except List<T> and Set<T> which are directly supported)

Names will be resolved during compilation of the hilang file into source code for the target language[[35]](#endnote-36)

### Reference



References are often a single name, or include their namespace separated by “.”. The name “Sparx.EA.Connector” is a reference where the namespace “Sparx.EA” is included. HiLang references are always fully qualified in the definition, but the generated code is wrapped within a namespace.

For example, the entity Sparx.EA.Element has a parent (that is also an entity), but only the key is stored with the child since the parent can be retrieved from hiperspace if needed.

@Versioned entity Sparx.EA.Element (Id : Int32) {Parent : Sparx.EA.Element};

### Inheritance



Elements can “inherit” from other elements, the definition will be expanded during generation so that the concrete element includes all the keys, values, and extensions of the base type. Inheritance is especially useful for ensuring that all concrete elements have the same shape as a base type. In future HiLang may generate base types as interfaces but the focus is on performance, so virtual properties are excluded.

### Views

An element can optionally be project one or more views that allow entities to be seen as a common view type (with enumeration from the views SetSpace<>). View can be declared as

* Alias  
  The view is delegated to a member of the entity (e.g. Price).
* Definition  
  Each member of the view is mapped to an expression that provides the members value.
* Implement  
  The names of the view members are mapped to members of the element that have the same names.

### Keys



An element key consists of one or more fields separated by commas. Each field can be a basic type, a value or a reference to an entity or a segment. When an Entity is extended with a segment, a reference to the owner is added to the start of the key. Keys can also be a list of elements, but this is discouraged because the key can become very long.

The notation uses open bracket “(“and close bracket “)” like a mathematical expression, where these are the values that are known before looking-up the element in Hiperspace.

#### Field



Each field has a text name followed by a colon and the Name of the datatype. The hash field is optional (as with elements), but recommended to ensure that changes to field names do not result in errors when opening a Hiperspace (with Meta-Model parameter)

### Values



Element values consists of one or more value separated by commas. Each value can be a basic type, a value type or a reference to an entity or a segment. When an Entity is extended with a segment, a reference to the owner is added to the start of the key. Keys can also be a list of elements, but this is discouraged because the key can become very long.

The notation uses open set “{“and close set “}” like a mathematical expression, where these are the values that are available from Hiperspace when you know key.

#### Value



Values either have a type or an expression that is evaluated when the element is bound to a SubSpace (if it does not have a value). The expression is used if the object does not have a value, in this example “Deleted = false” can be set prior to Bind() to tombstone the object: useful when CQRS[[36]](#endnote-37) uses different SubSpaces for load and read (for read, you’d want a Horizon filter to hide deleted elements, but would block Bind() if the element did not match the condition). “Valid = …” is an example where you’d want the Horizon filter to block write of invalid values).

Constraints: Set<Sparx.EA.ObjectConstraint>,

"Tombstone" Deleted = false,

"Check constraints" Valid = Name = null || ObjectType = null ||Package = null ? false : true

### Extents



Elements extent consists of one or more Extent separated by commas. Each Extent is either a relation to another element, or an expression. When an Entity is extended with a segment, a reference to the owner is added to the start of the concrete element key.

Extents are not stored with the element they extend but are navigable from as if they were part of the body. Extents add to immutable elements and avoid the need to version non-immutable ones.

The notation uses open collection “[“and close collection “]” like a mathematical vector/matrix, but commonly used in programming languages to indicate a collection.

#### Extent



Relations are either references to other entities, segments, or aspects (one of). In this example Attribute is a relation to another entity (using the self KeyRef<> property of the element), while ObjectMetric is a segment that inherits the key (as owner) from the parent element.

Attributes : Sparx.EA.Attribute (Element = self),

Metrics : Sparx.EA.ObjectMetric,

Balance = sum (Payments.Value)

Unlike value expressions, extent expressions are evaluated when needed, and most useful for aggregates of relational extensions of a related entity or segment. Functions can be added using directive, with basic aggregate functions built in[[37]](#endnote-38).

### Enums



In common with other Elements enums can optionally include documentation and properties, but do not have a hash code because they will never be stored directly within Hiperspace.

Enums have one or more enum values.



In common with {keys, values, extents} enum values can include documentation and properties. They must have a hash since the hash value is what is stored for the value as part of another element.

enum Status

{

Draft #1, /\* equvilent to protobuf \*/

Open #2, /\* Open = 2; but uses # to be \*/

Closed #3 /\* distinct from expressions \*/

};

## Directive



Directives pass instructions to the HiLang, current directive include:

|  |  |  |
| --- | --- | --- |
| Name | Description | Example |
| function | Declares a function that will be implemented externally.  Parameters are   1. {set, aggregate, unary} 2. Function name used in hilang source 3. The full name of the implementationThe first parameter is {set, aggregate, unary}, 4. Optional data type of the function | %function (aggregate, sum, Functions.Sum); |
| domain | Name of the DomainSpace that will be generated with SetSpace<> members for each of the Entity, Segment, aspect, view in the domain model. | ﻿%domain (Sparx); |
| import | Include another “.hilang” file in the generation | %import ("Element.hilang"); |

## Expression



HiLang expressions include all the normal arithmetic operations (“+” also applies to strings), brackets for ordering and conditional expressions using C/C++/C# notation.

Functions have a reference to the function name and several parameter reference to other members within the element.

### Boolean



Conditional expressions can include range, equality and Boolean AND (&&) and OR (||)

## Hash



The hash code is very important in HiLang because we use “Protocol Buffers” (Protobuf[[38]](#endnote-39)) for serialization of the Key (modified for search) and Value. Elements are serialized and deserialized using the hash number in place of the field/value name for efficiency.

{Entities, Segments, Aspects, Indexes} also have hash codes because they are fields in the envelop of the serialized buffer for key and value.

If an element or key/value does not have hash code, one is assigned starting with the highest unused value. It is best practice to always assign a hash code to elements, key, and values to ensure that an additional item is not assigned the same number as an item that was previously used by a field that was removed.

## Prelude

### Types

|  |  |
| --- | --- |
| Type | Notes |
| Int16 | Integers in the range ±32,768 |
| Int32 | Integers in the range ± 2,147,483,648 |
| Int64 | Integers in the range ± 9,223,372,036,854,775,808 |
| Decimal | Upto 28 digit precise real number |
| Double | Double precision floating point number |
| Single | Single precision floating point number |
| Half | Small floating point number for AI workloads using two bytes of storage |
| String | Text up to 2Gb in size |
| DateTime | Date and Time with millisecond precision |
| Timestamp | Date and Time with default value of Current DateTime |
| Boolean | True or false |
| Guid | Globally unique identifier |
| Identity | Guid initialized with a unique key |

### Collections

|  |  |
| --- | --- |
| Type | Notes |
| Set<> | Generic collection of unique items |
| List<> | Generic collection of ordered items |

### Functions

Functions can be declared in a HiLang file using the %function directive in addition to the built in functions

|  |  |  |
| --- | --- | --- |
| Function | Usage | definition |
| sum | Sum of collection . value “sum(Transactions.Cost)” | %function ( aggregate, sum, Functions.Sum); |
| avg | Average of collection . value “avg(Transactions.Cost)” | %function ( aggregate, avg, Functions.Avg); |
| Max | Maximum value of collection . value “max(Transactions.Cost)” | %function ( aggregate, max, Functions.Max); |
| Min | Minimum value of collection . value “min(Transactions.Cost)” | %function ( aggregate, min, Functions.Min); |
| Count | Count of values in collection . value “count(Transactions.Cost)” | %function ( aggregate, count, Functions.Count); |

## Graph Views

Two views are predefined for a SubSpaces to present the content of the SubSpace as a graph of Nodes and Edges between nodes. Every Element that maps the Node view is included in the Node SetSpace, while every Elements that maps the Edge view is included in the Edge SetSpace.

Graphs in Hiperspace are directed ***from*** an element and ***to*** another, the reverse of the edge can be performed in application code if needed.

Node contains a RefSet of Edges for ***Froms*** and ***Tos*** for all the edges that connect to other nodes. Nodes as are keyed by the “stringification” of the underlying element key. The domain SubSpace has a Get() function to retrieve the underlying object that provided the node. Navigation from Node to edge is possible as long as the Edge view represents the key to the element that provides the edge.

Additional nodes/edges can be represented through intermediate views. Views are not interfaces – a complex objects can be released from memory once the view is retrieved.

A screenshot of a computer

Description automatically generated

Most Element that provided a graph view will implement both Node and Edge views since Edge parameters must be Nodes, and Nodes without Edges gain nothing from a graph view.

entity Plan.Tasks.Task =

Node (SKey = SKey, Name = Name, TypeName = "Plan-Task"),

Edge (From = self, To = Project, TypeName = "Plan-Task", Name = Name),  
 Plan.Tasks.SubTask (Task = self, Parent = Parent)..;  
  
view Plan.Tasks.SubTask =

Edge (From = Task, To = Parent, TypeName = "Plan-Task-Part", Name =   
 Task.Name) ( Task : Plan.Tasks.Task ) { Parent : Plan.Tasks.Task};

|  |  |
| --- | --- |
| Code | Notes |
| "node in a graph view of data"  view Node  (  SKey : String #1,  TypeName : String #2  )  {  Name : String #3,  }  [  From : Edge,  To : Edge  ]; | Skey is the stringified (using base-64) of element.  Skey and TypeName provide the key to the Node since an Element might represent several nodes.  Name is the only property. Properties retrieved by loading the underlying element from the SubSpace.  Node has extensions to validate to the set of edges, when the view matches with the elements key. |
| "edge between nodes"  view Edge  (  From : Node #2,  To : Node #3,  TypeName : String #4  )  {  Name : String #5,  }; | An edge is a connection between two Nodes  TypeName is part of the key, since there may be several links between the nodes for different roles.  Edge does not have extensions- navigation from members is built-in as Node is reference type. |

# Index

AsAt 7, 18, 27

aspect 20, 26, 34

attribute 7

blocked 8

Boolean 36

concurrency 8

database 7

datasets 9

definition 11, 23, 24, 25, 26, 28, 29

deleted 7

Directive 34

Documentation 26, 27

domain 34, 43

Edge 8

Elements 7

entity 8, 11, 22, 25, 26, 28, 29, 30, 31, 32, 43

Enums 33

evolution 8

Extent 32

Extents 32

filter 7

function 20, 34, 35

Graph 8, 10

Hash 36

historical 9

history 7

horizon 7

Immutability 8

immutable 7

import 27, 34

IndexPath 17, 22

Inheritance 29

Key 16, 36

KeyPath 21

KeyRef 16, 20, 32

Keys 6, 30, 31

locking 7

Mutability 7

Name 11, 28, 30, 31, 34

Node 8, 10, 26, 43

optimizer 6

Property 27

Reference 28

RefSet 16, 20

RefSingle 20

Sargable 43

Schema 8

segment 26, 30, 31, 32, 43

SetSpace 16, 17, 19, 20, 25, 26, 29, 34

space 7

SQL 8, 10, 23

SubSpace 5, 7, 9, 16, 17, 18, 19, 20, 26, 31

tombstone 7

tombstoning 7

type 10, 16, 19, 25, 26, 28, 29, 30, 31

value 8, 10, 11, 14, 15, 16, 25, 27, 29, 30, 31, 32, 33, 36

Value 16, 20, 31, 32, 36

Values 6, 25, 31

versioned 7

versions 7

view 7, 8, 9, 10, 26, 29, 34, 43

Views 8, 26, 29

1. https://protobuf.dev/ [↑](#endnote-ref-2)
2. https://github.com/channell/Hiperspace [↑](#endnote-ref-3)
3. e.g. owner of a segment [↑](#endnote-ref-4)
4. <http://openssd-project.org/kvssd/> [↑](#endnote-ref-5)
5. <https://en.wikipedia.org/wiki/Temporal_database> [↑](#endnote-ref-6)
6. Environmental, Social Governance [↑](#endnote-ref-7)
7. <https://en.wikipedia.org/wiki/Fundamental_Review_of_the_Trading_Book> [↑](#endnote-ref-8)
8. <https://en.wikipedia.org/wiki/Data_lake> [↑](#endnote-ref-9)
9. See example of and entity projecting a graph view at <https://github.com/channell/Hiperspace/blob/master/examples/Hiperspace.Sparx/Hilang/Element.hilang> [↑](#endnote-ref-10)
10. <https://en.wikipedia.org/wiki/WebAssembly> [↑](#endnote-ref-11)
11. <https://github.com/channell/Hiperspace/tree/master/examples/Plan/Plan> [↑](#endnote-ref-12)
12. <https://en.wikipedia.org/wiki/Snappy_(compression)> [↑](#endnote-ref-13)
13. <https://www.intel.com/content/www/us/en/products/docs/memory-storage/optane-persistent-memory/overview.html> [↑](#endnote-ref-14)
14. <https://en.wikipedia.org/wiki/IBM_Future_Systems_project> [↑](#endnote-ref-15)
15. <https://en.wikipedia.org/wiki/Content_Addressable_File_Store> [↑](#endnote-ref-16)
16. <https://en.wikipedia.org/wiki/GNU_General_Public_License> [↑](#endnote-ref-17)
17. <https://www.nuget.org/packages/Hiperspace> [↑](#endnote-ref-18)
18. <https://www.nuget.org/packages/Hiperspace.Rocks> [↑](#endnote-ref-19)
19. <https://github.com/facebook/rocksdb/> [↑](#endnote-ref-20)
20. <https://www.speedb.io/> [↑](#endnote-ref-21)
21. <https://semiconductor.samsung.com/solutions/technology/pim/> [↑](#endnote-ref-22)
22. <https://en.wikipedia.org/wiki/High_Bandwidth_Memory> [↑](#endnote-ref-23)
23. <https://github.com/rockset/rocksdb-cloud> [↑](#endnote-ref-24)
24. <https://sparxsystems.com/> [↑](#endnote-ref-25)
25. Persistence tools like Entity Framework use proxies to track changes and provide navigation, but normally require the creation of wrapper objects in addition to domain objects. [↑](#endnote-ref-26)
26. Which never changes, as they are immutable. [↑](#endnote-ref-27)
27. Only the domain knows the full set of elements that can viewed as a Node or Edge. [↑](#endnote-ref-28)
28. Language Integrated Query https://learn.microsoft.com/en-us/dotnet/csharp/linq/ [↑](#endnote-ref-29)
29. Sargable is an old database term for segment argument and indicates that it is indexed <https://en.wikipedia.org/wiki/Sargable> [↑](#endnote-ref-30)
30. <https://github.com/channell/Hiperspace/tree/master/examples/Hiperspace.Sparx> [↑](#endnote-ref-31)
31. <https://www.nuget.org/packages/HiLang/> [↑](#endnote-ref-32)
32. <https://en.wikipedia.org/wiki/Recursive_transition_network> [↑](#endnote-ref-33)
33. <https://en.wikipedia.org/wiki/Abstract_syntax_tree> [↑](#endnote-ref-34)
34. <https://en.wikipedia.org/wiki/XML_Metadata_Interchange> [↑](#endnote-ref-35)
35. Initially .hilang files are generated to C#, but will (in future) include {F#, Java, C++/Rust} in future. [↑](#endnote-ref-36)
36. <https://learn.microsoft.com/en-us/azure/architecture/patterns/cqrs> [↑](#endnote-ref-37)
37. <https://github.com/channell/Hiperspace/blob/master/src/Hiperspace/Functions.cs> [↑](#endnote-ref-38)
38. <https://protobuf.dev/> [↑](#endnote-ref-39)