Databases

```
as_error err;
as_config config;
as_config_init(&config);

config.hosts[0] = { .addr = "127.0.0.1", .port = 3000 };

aerospike as;
aerospike_init(&as, &config);

aerospike_connect(&as, &err);

as_key key;
as_key_init(&key, "test", "demo_set", "test_key");

as_record rec;
as_record_inita(&rec, 2);
as_record_set_int64(&rec, "test-bin-1", 1234);
as_record_set_str(&rec, "test-bin-2", "test-bin-2-data");

aerospike_key_put(&as, &err, NULL, &key, &rec);

aerospike_close(&as);
aerospike_destroy(&as);
```

Initialize Record Data

The aerospike_key_put() operation requires an initialized record, which contains the bins to be stored in the cluster. Here, we initialize a record on the stack with 2 bins. The first bin is called "test-bin-1", and is an integer type. The second bin is called "test-bin-2", and is of type string.

```
as_record rec;
as_record_inita(&rec, 2);
as_record_set_int64(&rec, "test-bin-1", 1234);
as_record_set_str(&rec, "test-bin-2", "test-bin-2-data");
```

Initialize Key

When writing records, you need to identify the record in the database via a key. Below is the key we create for the record above. The value used for the key is a string called "test-key", to be stored in the namespace "test", and within the set "test-set". It is possible to have other data types, such as integer or blob, to be used as key.

```
as_key key;
as_key_init_str(&key, "test", "test-set", "test-key");
```

Write to Database
With the key, we can now write the record into the database:

```
if (aerospike_key_put(&as, &err, NULL, &key, rec) != AEROSPIKE_OK) {
    fprintf(stderr, "err(%d) %s at [%s:%d]\n", err.code, err.message, err.file, err.line);
}
```

Clean up Resource
When you no longer require a record, you should release it and associated resources.

```
as_record_destroy(rec);
```

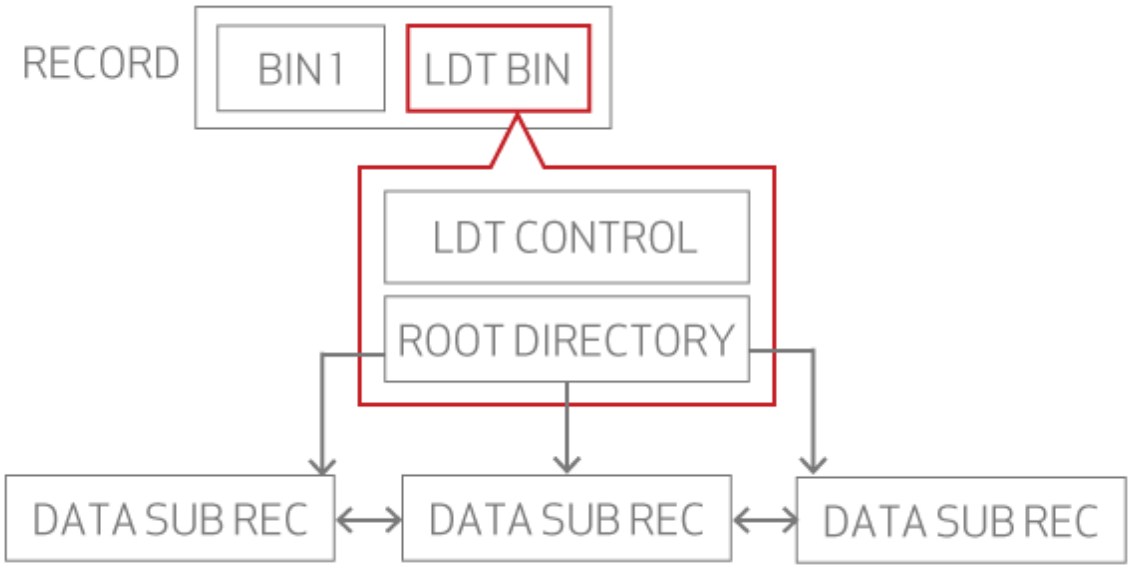
The following shows an array of 2 bins named "test-bin-1" and "test-bin-3", which we want to read from the database. The array is NULL-terminated, telling the read operation that there are no more bin names.

```
as_record* p_rec = NULL;
static const char* bins_1_3[] = { "test-bin-1", "test-bin-3", NULL };

if (aerospike_key_select(&as, &err, NULL, &key, bins_1_3, &p_rec) != AEROSPIKE_OK) {
    fprintf(stderr, "err(%d) %s at [%s:%d]\n", err.code, err.message, err.file, err.line);
}

as_record_destroy(rec);
```

When you no longer require the record, be sure to release it via as_record_destroy().

Objects in a LDT are not stored contiguously with the associated record, but instead are split into sub-records (with sizes ranging roughly from 2kb to 200Kb, upper bound determined by write block size).

Large Ordered List Operations
Physically, the Large Orderd List is implemented with a B+ Tree, where the inner tree nodes and leaves are sub-records. The root of the B+ Tree resides in the parent record.



The B+
-tree is essentially a mechanism for
managing a sorted array-based list, where the list is broken into chunks.